OBJEKTUMORIENTÁLT SZOFTVERTERVEZÉS

2020 – teljes tananyag

Szerkesztette és
jegyzetbe rendezte:
Fábián Csenge

Tartalom

Objektumorientált tervezési elvek

OO fogalmak

osztály

- = típus
- metódusok/ tagfüggvények: a viselkedést definiálják
- mező/ attribútum/ tagváltozó: állapotot írják le

objektum

- egy osztály példánya

statikus tag

- osztály szintű
- minden példány ugyanazt az értéket látja
- UML diagramon aláhúzással jelöljük őket
- akkor is elérhetőek, és meghívhatók, ha még nem létezik objektum az adott osztályból
- a statikus metódusok nem is tudják közvetlenül elérni a példány tagokat

példány tag

- objektum szintű
- minden példányban más értéket vehetnek el
- a this vagy self pointer az aktuális objektumot reprezentálja
    - ezen a mutatón keresztül érhetőek el az objektum saját tagváltozói és tagfüggvényei

asszociáció

- erős kapcsolat, függvényhívásokon túl is megmarad
- a szemközti osztály egy példányát vagy példányait egy attribútumban tároljuk
- UML jelölés: tároló ⟶ tárolt
- még erősebb fajtája a tartalmazás/kompozíció
    - a tartalmazott objektumok együtt élnek és együtt halnak a tartalmazó objektummal
    - egy objektumot egyszerre csak egy tartalmazó objektum tartalmazhat
        - tartalmazó objektum másolásakor a tartalmazott objektumok is lemásolódnak
        - a tartalmazó objektum megszűnésekor a tartalmazott objektumok is megszűnnek
    - UML jelölés: tartalmazott ◆⟶ tartalmazó

függőség

- a kliens függ a szervertől, tehát a kliens használja a szervert
- a függőség azért gyengébb kapcsolat, mint az asszociáció, mert a kliens nem tárolja attribútumban a szervert, csak ideiglenesen használja azt

- például: a kliens függvényének attribútuma vagy visszatérési értéke a szerver objektum, a kliens létre hoz egy új szervert
- UML jelölés: kliens - - - ≫ szerver

Kapcsolatok erőssége

- fontos megjegyezni a kapcsolatok erősségét, mert az erősebb kapcsolat mindig implikálja a gyengébbet
- vagyis, ha két osztály között asszociáció van, akkor függenek is egymástól
- ha egy osztály tartalmaz egy másikat, akkor közöttük egyben asszociáció is van

interfész

- két rendszer vagy két komponens mindig egy interfészen keresztül kapcsolódik egymáshoz
- egy interfész azt a függvényhalmazt, szolgáltatáshalmazt definiálja, amit az adott interfészt megvalósítható rendszeren meghívhatunk
- az interfész tehát valójában egy névvel ellátott függvényhalmaz
- elvárt viselkedés tartozik hozzá, amit dokumentálni kell
- csak a függvények fejlécét és a tőlük elvárt viselkedést definiálják, a tényleges implementációt az osztály biztosítja
    - emiatt interfészek nem példányosíthatók, mert önmagukban nem rendelkeznek semmilyen funkcionalitással

implementáció

- az interfészeket az osztályok implementálják
- UML jelölés: osztály ⋯▷ interfész

osztály interfésze

- egy osztály interfésze a rajta hívható publikus függvények halmaza (nem az általa implementált interfészekkel egyenlő!)
- az osztály példányaival ezeken a függvényeken keresztül lehet kommunikálni

öröklődés

- célja az ősosztályban definiált viselkedés újra hasznosítása és kibővítése
- UML jelölés: gyerek —▷ szülő

virtuális függvény

- az ősben definiált virtuális függvényeket felül lehet írni a leszármazott osztályban
- a statikus függvények nem lehetnek virtuálisak, hiszen azoknak példány nélkül is meghívhatónak kell lenniük
- UML jelölés: nincs külön jelölése, mert az UML feltételezi, hogy minden tagfüggvény alapból virtuális
    - de ha egy leszármazott egy virtuális függvényt felülír, akkor az UML diagramban ezt a függvényt a leszármazottban megismételjük

polimorfizmus

- ha egy kliens az ősön keresztül hív meg egy virtuális függvényt, akkor valójában a leszármazottban felülírt változat fog lefutni

    - ezt hívjuk polimorfizmusnak, és ez az objektumorientált tervezés fő erőssége
- a kliensnek, vagyis a hívónak nem szabad arról tudnia, hogy szerver, vagyis a meghívott objektum valójában az ősosztálynak, vagy annak valamilyen leszármazottjának példánya
- a kliens számára az ősnek és leszármazottaknak is ugyanazt az elvárt viselkedést kell biztosítaniuk, így a kliens mindenkit egységesen, az ősön keresztül tud elérni

absztrakt függvény

- olyan virtuális függvény, amely nem rendelkezik alap implementációval
- UML jelölés: dőlt betű

absztrakt osztály

- olyan osztályok, amelyek legalább egy absztrakt függvény tartalmaznak
- nem példányosíthatók, mert az absztrakt függvényükben nincs viselkedés, ami le tudna futni
- egy osztályt akkor is absztrakttá tehetünk, ha nincsenek absztrakt függvényei (ezek sem példányosíthatóak)
- UML jelölés: dőlt betű

Konkrét osztályok

- nem absztrakt osztályok
- mindig példányosíthatók
- kötelesek implementálni azokat a függvényeket, amelyek valamely ős interfészben vannak definiálva, vagy valamely ősben absztraktok, és azokat valamely korábbi másik ős még nem implementálta
- ha egyértelmű, hogy mely függvényeket kell implementálni, akkor ezeket az UML ábrán nem kell megismételni

Láthatóság (UML)

- privát (-)
    - az adott mezőhöz vagy metódushoz csak az őt tartalmazó osztály fér hozzá
- protected (#)
    - az adott mezőhöz vagy metódushoz csak az őt tartalmazó osztály és annak leszármazottjai férnek hozzá
- public (+)
    - mindenki hozzáfér, akinek az adott osztályhoz hozzáférése van
- package (~)
    - adott csomagon belül láthatók
- fontos megjegyezni, hogy a láthatóságok jelentése egyes programnyelvekben eltérhet az UML szemantikától
    - például C++-ben nem létezik package láthatóság, de a működését lehet szimulálni a friend kulcsszó segítségével
    - C#-ban pedig a package helyett az internal láthatóság érhető el, azonban ez nem a névtéren belüli láthatóságot jelenti, hanem az assembly-n belülii láthatóságot

absztrakció

- a világból csak releváns dolgokat modellezünk
- ami a szoftver működése szempontjából irreleváns, azt hagyjuk figyelmen kívül
- például, ha a Naprendszer bolygóinak mozgását szeretnénk modellezni, csak a gravitációs törvények relevánsak, a kvantummechanika törvényeit elhanyagolhatjuk

osztályozás

- az egymáshoz hasonló objektumokat egységesen, ugyanolyan osztállyal modellezzük
- ez az osztály írja le a közös viselkedést és a közös állapotot
- például a Naprendszer bolygói ugyanúgy viselkednek a gravitációs törvények hatására, ugyanolyan tulajdonságaik vannak (tömeg, sebesség), így ezeket lehet egy közös, 'Bolygó' osztállyal modellezni

egységbezárás (= adatrejtés)

- a külvilág csak azt láthassa az objektum állapotából, amit feltétlenül szükséges
- következmény: célszerű minden attribútumot privát láthatóságúnak felvenni, a hozzáférést pedig csak publikus vagy protected láthatóságú függvényeken keresztül megengedni - ezek tudják garantálni az állapot konzisztenciáját (pl.: hogy adott tartományba essen)

öröklés

- a hasonló viselkedésű osztályok közös tulajdonságát egy közös ősosztályban írjuk le
- a leszármazottak ezt a közös viselkedést megöröklik, és szükség szerint kiegészítik
- az ősosztálytól elvárt viselkedést a leszármazottak nem sérthetik meg
- az öröklés mindig a viselkedés újrahasznosításáról szól, sosem az attribútumok, adatok újrahasznosításáról

kohézió

- annak mértéke, hogy egy modul vagy osztály tagjai mennyire tartoznak össze, mennyire erős a kapcsolat a tagok között
- ha az osztály tagváltozói és tagfüggvényei olyan részhalmazokra bonthatók, melyek között nincs kapcsolat, akkor az osztályban nincs meg a kohézió, a részhalmazokból önálló osztályokat kellene alkotni
- a szoftver karbantarthatósága szempontjából az erős kohézió az előnyös, mert ilyenkor az egy funkcióhoz tartozó dolgok egy helyre összpontosulnak, a szükséges változtatás helye lokalizált

csatolás

- annak a mértéke, hogy az egyes modulok, komponensek, osztályok vagy függvények mennyire függenek egymástól
- ha túl sok ez a keresztfüggőség, akkor egy-egy változtatás a szoftver sok más részére kihatással van
- a szoftver karbantarthatósága szempontjából az alacsony (más néven laza) csatolás az előnyös

OO tervezési elvek

Követelmények változása

- a követelmények gyakran változnak, mert a megrendelő nem tudja előre pontosan megfogalmazni, hogy mit szeretne, fejlesztés közben jönnek az újabb ötletei, amikor már látja a terméket működés közben
- vannak olyan szoftverek, amelyek folyamatosan fejlődnek, újabb és újabb funkciókkal bővülnek
- a jól megtervezett szoftverben a változtatásokat könnyű véghezvinni

Rosszul tervezett szoftver:

- merev (sok helyen kell változtatni)
- törékeny (nem várt részeken okoz hibát a változtatás)
- nem újrahasznosítható (nehéz átemelni az egyes komponenseket, mert a komponensek túlságosan összegabalyodnak egymással)

jól tervezett szoftver

- részek közötti függőség csökkentése
- a függőségeket ne a gyakran változó, problémás részek felé irányítsuk
- tervezői döntések dokumentációja, változtatások dokumentálása
    - ez a későbbi változtatásoknál elengedhetetlen lesz, hogy azok ne menjenek szembe az eredeti tervezési döntésekkel
- gondolni kell a lehetséges változtatásokra
- a kis valószínűségű változtatásokra felesleges felkészülni és így "túltervezni a rendszert"
    - YAGN = You Ain't Gonna Need
    - a túltervezett szoftver túl sok overhead-el jár

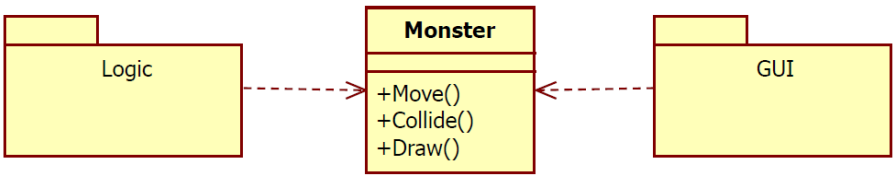
SOLID OO tervezési elvek

SOLID elvek

- A jó OO tervezés öt alapelve (Robert C. Martin)
    1. Single Responsibility Principle (egyetlen felelősség elve)
    2. Open-Closed Principle (nyitottság-zártság elve)
    3. Liskov Subtitution Principle (Liskov-féle helyettesítés elve)
    4. Interface Segregation Principle (interfészek szétválasztásának elve)
    5. Dependency Inversion Principle (függőségek megfordításának elve)
- cél
    - karbantarthatóság
    - bővíthetőség
    - függőségek csökkentése

Single Responsibility Principle (egyetlen felelősség elve)

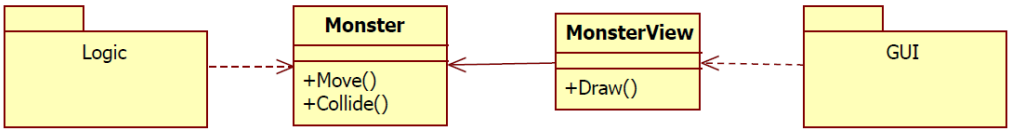
- probléma: egy osztályban 2 féle felelősség is keveredik, így bármelyik megváltoztatása a másikat is érinti



- SRP: *"egy osztálynak csak egyetlen oka lenyen a változásra"*
- ha ez nem teljesül, az osztályt szét kell választani a felelősségek mentén
  - implementációs szinten: több osztályt készítünk belőle
  - ha az osztály implementációja túl bonyolult: osztály interfészeinek szétválasztása (felelősségenként egy-egy), a kliensek csak a számukra érdekes interfészektől függenek (interfészek szétválasztásának elve)

Pull jellegű megoldás

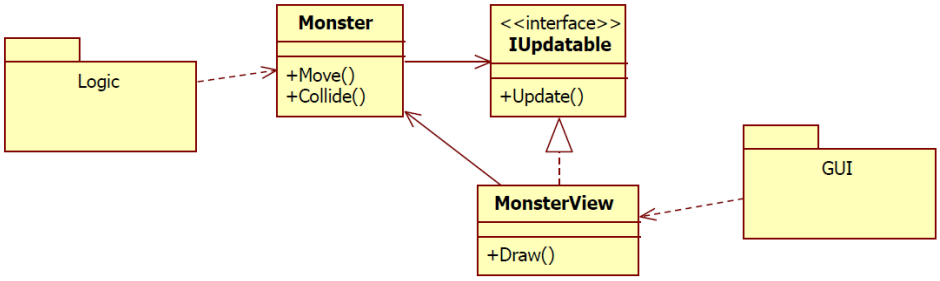
- a logikai és grafikai rész különválasztása
- a grafikai rész függ a logikai résztől (rajzolás során mindig lekérdezzük a modell állapotát)



- előnye
  - a grafikai rész általában gyakrabban változik, mint a logikai, és ennek a változásnak nincs hatása a rendszer többi részére
- hátránya
  - gyakori újrarajzolás, melyre akkor is sor kerül, ha modell állapotában nem történt változás

Push jellegű megoldás

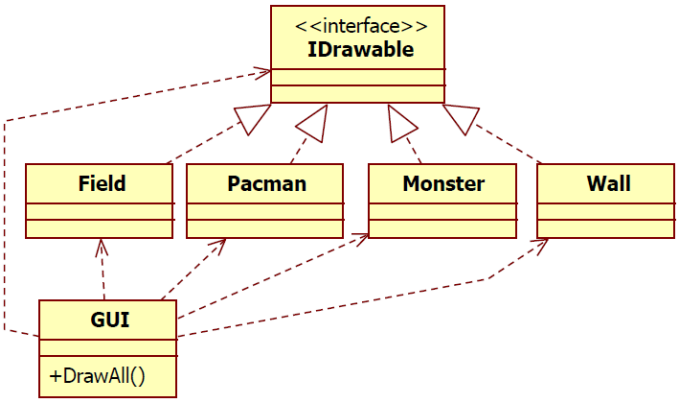
- a logikai és grafikai rész különválasztása + IUpdatable interfész
- a modell értesíti a grafikát, ha változás történt
- ilyenkor a grafika lekérdezi a modell új állapotát és az alapján frissíti a felületet
- ehhez a modellnek ismernie kell a grafikát, hogy tudja azt értesíteni - de ezt a kapcsolatot jól le kell választani, nehogy a grafikai változtatások visszahassanak a modellre
  - a modellben definiált IUpdatable interfészt a grafikus komponensek implementálják, és a modell csak ezen az interfészen keresztül látja a grafikus interfészeket
- így a logika nem függ a grafikus résztől, ezért a grafikus rész megváltozása nem hat modellre



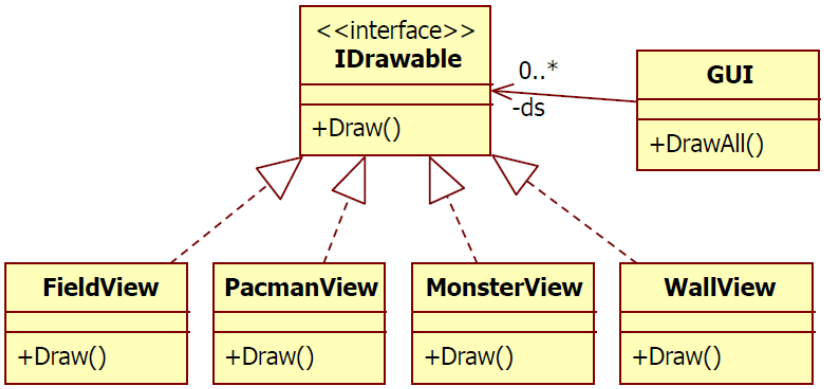
- előnye
  - csak változáskor történik újrarajzolás

Open-Closed Principle (nyitottság-zártság elve)

- probléma: ha egy új szereplő jelenik meg a programban, a szoftvert sok helyen kell változtatni



- OCP: *"a szoftver részeinek nyitottnak kell lennie a kiterjesztésre, de zártnak a módosításra"*
- tehát fel kell készülni a változásokra, a szoftver úgy kell tudunk kibővíteni, hogy a meglévő részekhez már nem nyúlunk hozzá
- cél: új kód írásával bővítsük a funkcionalitást, ne a meglévő átírásával
- funkcionalitás bővítésének módja: új osztályok, virtuális függvények felülírása, polimorfizmus, delegálás
- OCP megoldás:



  - minden nézet osztály implementálja az IDrawable interfész draw függvényét, és a GUI DrawAll() függvénye ezt a Draw() függvényt hívja mindenkin
  - egy új szereplőnek ugyanezt az IDrawable interfészt kell implementálnia, a többi osztály kódjához nem kell hozzányúlni

Liskov Subtitution Principle (Liskov-féle helyettesítés elve)

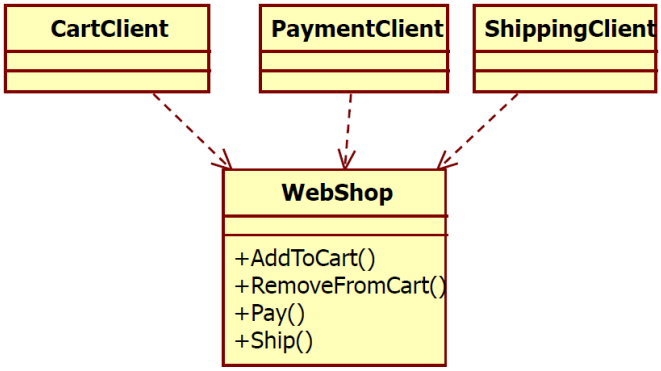
- LSP: *"A leszármazottaknak behelyettesíthetőnek kell lenniük az ősbe: nem sérthetik meg az őstől elvárt viselkedést"*
- az elv arra épül, hogy a kliens az ős osztályon keresztül használja szervert, és így van egy elvárt viselkedés, amire az őstől a kliens számít
- a szerver lehet ugyan leszármazottja ennek az ősnek, de nem léphet ki azokból a keretekből, amikről a kliens tud
- ha ez mégis megtörténne, a kliens kénytelen lenne rákérdezni a szerver konkrét típusára, hogy speciális estként le tudja kezelni
- azonban ezáltal a klienst az OCP elv megsértésére kényszerítjük

Design-byContract (DbC)

- Liskov-elv betartását segíti, ha szerződéseket határozunk meg
- a leszármazott nem sértheti meg az ős szerződését
- Szerződés
  - előfeltételek: mit várunk el a hívótól
  - utófeltételek: mit garantálunk a hívó számára
  - invariánsok: mik azok a dolgok, amelyek állandóak
- Szerződés példa: pop művelet
  - előfeltétel: legalább egy elem legyen a veremben
  - utófeltétel: a hívó a legfelső elemet kapja vissza, a hívás végén eggyel kevesebb elem marad a veremben, és a megmaradt elemek sorrendje nem változik
- DbC szabályok a leszármazottakban
  - előfeltétel ugyanolyan vagy gyengíthető
  - utófeltétel ugyanolyan vagy erősíthető
  - invariáns ugyanolyan vagy erősíthető
- például:
  - ha az ősben egy előfeltétel az volt, hogy csak nemnegatív egész számokat fogad el bemenő feltételként, akkor a leszármazott ezt gyengítheti, és elfogadhat például minden egész számot, de nem erősítheti a feltételt azzal, hogy csak a pozitív egész számokat fogadja el, hiszen a kliens az ősön keresztül nullát is beküldhet argumentumként

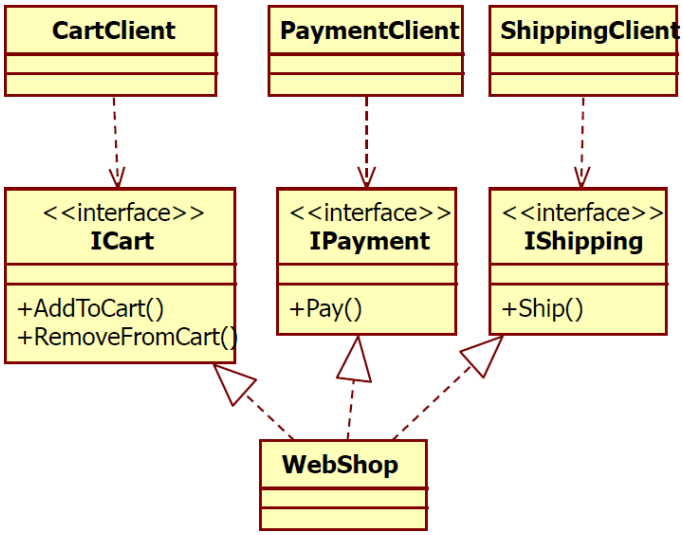
Interface Segregation Principle (interfészek szétválasztásának elve)

- probléma: a WebShop osztály sérti a SRP elvet, mert nagyon sok felelőssége van, ráadásul az egyes kliensek csak egy-egy részhalmazát használják a metódusoknak
  - a Webshop osztályt szét kellene darabolni több osztályra, azonban, ha az implementációja túl bonyolult, akkor ezt nagyon nehéz megtenni
  - ilyenkor valamilyen más megoldást kell találni

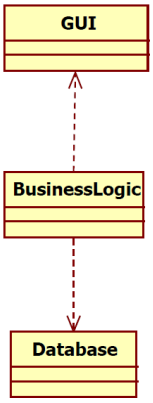


- ISP: *"a klienseket nem kötelezhetjük arra, hogy olyan metódusoktól függjenek, amelyeket nem használnak"*
- az ISP tehát elfogadja, hogy bizonyos esetekben egy-egy osztály túlságosan nagyra nőhet, és ezáltal a külső interfésze nem kohézív
- azonban a klienseket nem terhelhetjük azokkal a metódusokkal, amelyeket ők nem használnak, mert így a szerver osztályban történő, de őket nem érintő változások is hatással vannak rájuk
- a cél tehát az, hogy a kliensek csak azoktól a metódusoktól függjenek, amelyeket ténylegesen használnak

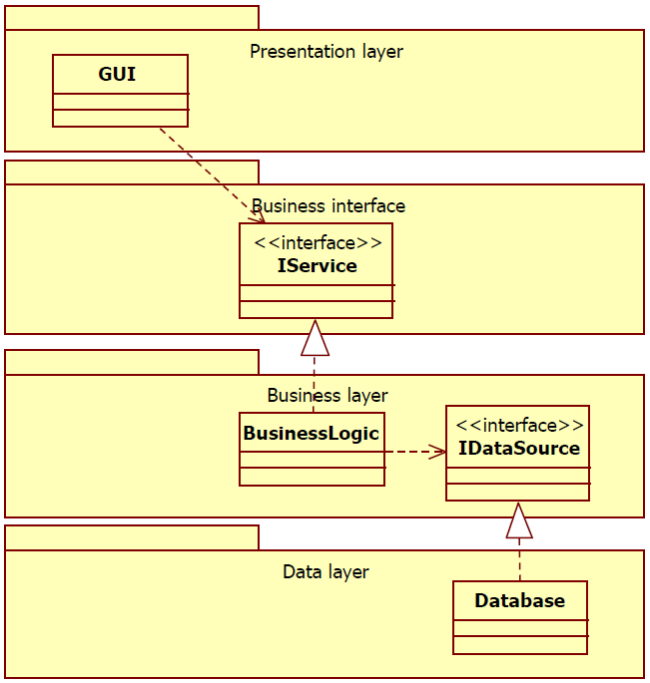
- a szerver interfészét fel kell darabolni kohézív részekre, ezeket az interfészeket a szerver mind implementálja, a kliensek azonban csak a számukra érdekes interfészektől függjenek
- így a kliensek le vannak választva a számukra érdektelen funkcióktól, és ezáltal a többi klienstől is függetlenné válnak
- megoldás: minden egyes kliensre definiáljuk azt az interfészt, amely számára releváns, a szerver pedig implementálja az összes interfészt
    - a szerver belső implementációján így módosítani nem kell, mégis a kliensek csak a számukra érdekes részeket látják, csak azoktól függenek



Dependency Inversion Principle (függőségek megfordításának elve)



- probléma: az üzleti logika függ a GUI komponenstől és az adatbázis komponenstől is
    - A GUI komponenstől való függés azért baj, mert a GUI gyakran változik, így az üzleti logikát is gyakran kell változtatni, de legalább újra kell tesztelni
    - az adatbázistól való függés azért rossz, mert az adatbázis elérése alacsony szintű művelet. Ha az adatbázist lecserélnénk egy másik gyártóéra, vagy relációs helyett dokumentum alapúra, akkor az üzleti logikai komponenst is át kellene írni
- DIP: *"Magas szintű modulok ne függjenek alacsony szintű moduloktól: mindketten absztrakcióktól függjenek."*
- az absztrakciók nem függhetnek a részletektől, a részleteknek kell függeniük az absztrakcióktól
- a magasabb szintű modulok definiálnak egy interfészt, amiben leírják, hogy mit várnak el az alacsonyabb szintű moduloktól
- az alacsonyabb szintű modulok ezt az interfészt implementálják
- így mind a magasabb és mind az alacsonyabb szintű modul ettől az interfésztől (absztrakciótól) függ



- a DIP arra is használható, hogy a függőség irányát megfordítsuk

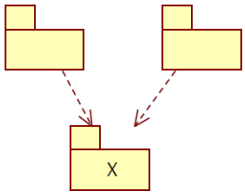
További OO tervezési elvek

1. Stable Dependencies Principle (stabil függőségek elve)
2. Stable Abstractions Principle (stabil absztrakciók elve)
3. Acyclic Dependencies Principle (aciklikus függőségek elve)
4. Don't Reapeat Yourself (ne ismételjük magunkat)
5. Single Choice Principle/ Single Point Control (egyetlen helyen történő esetszétválasztás elve)
6. Tell, Don't Ask (mondj, ne kérdezz)
7. Law of Demeter (Demeter törvénye)
8. Common Closure Principle (együtt változó osztályok elve)
9. Common Reuse Principle (közös újrahasznosítás elve)
10. Release Reause Equivalency Principle (kiadott komponensek újrahasznosításának elve)

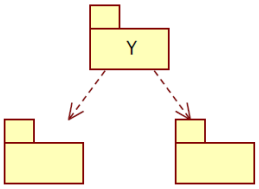
Stable Dependencies Principle (stabil függőségek elve)

- SDP: *"Mindig a stabilitás felé mutasson a függőség"*
- a stabilitás mértékét az határozza meg, hogy mekkora munka egy változtatás megvalósítása (minél nagyobb munka, annál stabilabb)
- a szoftvert úgy kell feldarabolni, hogy a ritkán változó stabil részektől függjenek a gyakran változó instabil részek
    - ennek felel meg például az is, hogy mindig a nézet függjön a modelltől, és ne fordítva
- az ábrán azt látjuk, hogy az X csomagtól két másik csomag is függ

    - ez egy jó ok arra, hogy az X-hez ne nagyon nyúljunk hozzá
    - mivel az X nem függ senkitől a rendszer többi részének megváltozása nincs hatással rá
    - az X tehát egy stabil csomag



- az Y csomag két másik csomagtól is függ, azonban tőle nem függ senki
    - ez azt jelenti, hogy Y bármikor megváltoztatható, a változásnak nem lesz másra hatása
    - azonban, ha az Y függőségei közül bármelyik módosul, az az Y-t is érinteni fogja
    - az Y tehát egy instabil csomag

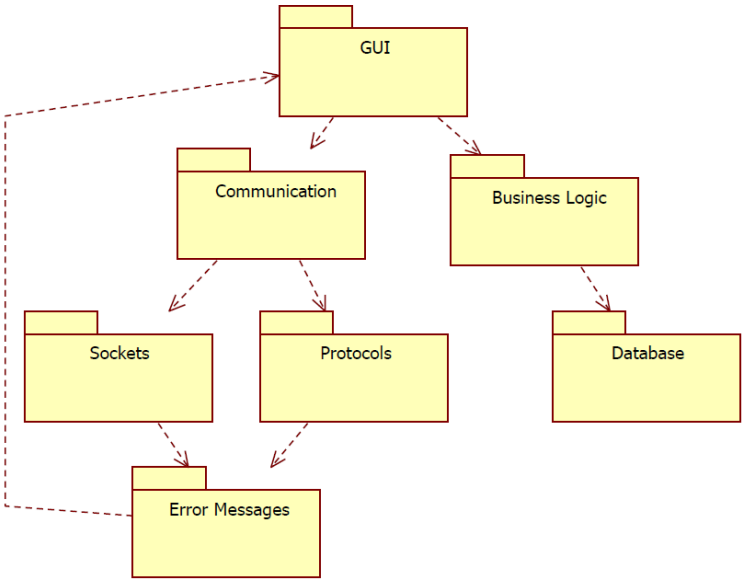


Stable Abstractions Principle (stabil absztrakciók elve)

- SAP: *"A stabil csomagok absztrakt csomagok legyenek"*
- attól még, hogy egy csomag stabil, és nehéz rajta változtatni, nem jelenti azt, hogy nehéz kiterjeszteni
- a jó megoldás az, hogy a stabil csomagok absztraktak, amiket könnyű kiterjeszteni, és erre épülnek az instabil csomagok, amiket könnyű megváltoztatni
- ez a Dependency Inversion Principle egy másik formája

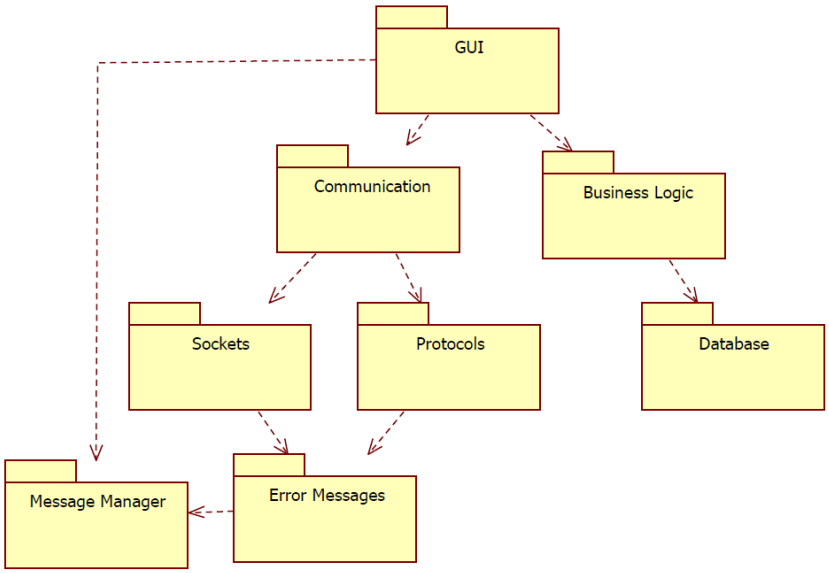
Acyclic Dependencies Principle (aciklikus függőségek elve)

- körkörös függőség problémája:
    - ha egy modul/csomag/komponens megváltozik, akkor újra kell fordítani, és az a tőle függő komponensekre is hatással van, mert lehet, hogy újra kell írni őket, de minimum újra kell őket tesztelni.
    - Ha körkörös függőség alakul ki, akkor ez minden, a körbe tartozó részre igaz.
    - Ez túlságosan nagy fejlesztési overheadet jelent.

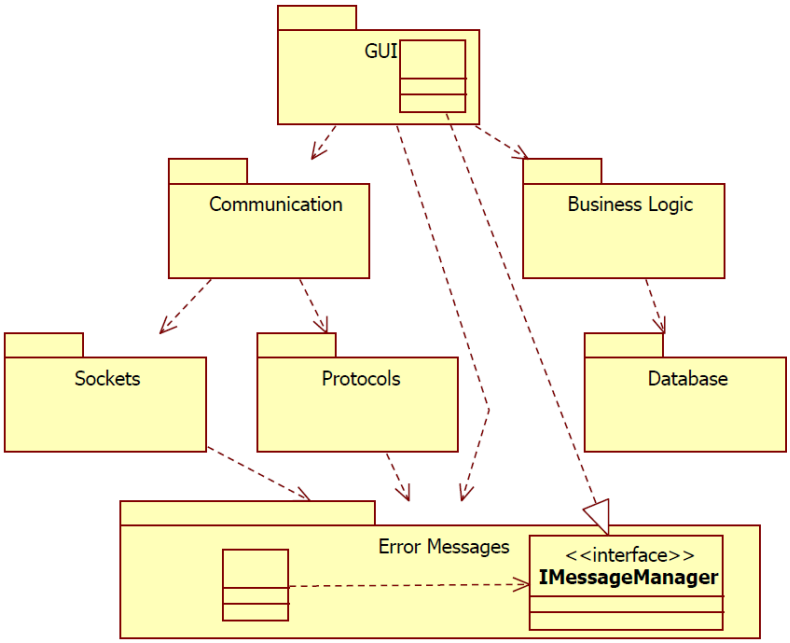


- ADP: *"A modulok, csomagok, illetve komponensek között ne legyen körkörös függőség"*
- ezáltal a változtatások hatására a szoftver minél kisebb részét kell újra fordítani és újra tesztelni

- ha mégis körkörös függőséget találunk, annak feloldására egy lehetséges megoldás lehet egy új csomag bevezetése:



- egy másik lehetőség a DIP segítségével ez egyik függőség irányának megfordítása:



Don't Reapeat Yourself (ne ismételjük magunkat)

- DRY: *"Ne ismételjük a kódot: minden tudás egyetlen egyértelmű helyen jelenjen meg a kódban!"*
- az ismétléssel az is a probléma, hogy ha az ismétlésbe hiba kerül, akkor azt minden helyen javítani kell. Ez sok munka, illetve könnyen kihagyható egy-egy eset
- ha fejlesztés közben lenyomjuk a Ctrl+C billentnyűkombinációt, akkor gondoljunk arra, hogy ebből a kódrészből inkább egy függvényt kellene készítenünk
- pl.: a lock műveletet a kliensek helyett a szerverre kell bízni, így a hívó objektumoknak nem kell kódot ismételniük
    - így sokkal tisztább, és egyben biztonságosabb is lesz a kliensek kódja

Single Choice Principle/ Single Point Control (egyetlen helyen történő esetszétválasztás elve)

- SCP: *"Valahányszor egy rendszernek sok különböző esetet kell megkülönböztetnie, az esetszétválasztás legfeljebb egyetlen, jól megválasztott helyen szerepeljen a kódban"*
- DRY és OCP következménye
- például a szoftver többnyelvűségét támogató erőforrásfájlok éppen ennek az elvnek a betartását segítik: a szoftverben található valamennyi szöveg, amely a felhasználói felületen megjelenhet, ebben a nyelvi erőforrásfájlban van összegyűjtve.
    - Így csak ezen az egy helyen kell felvenni az újabb eseteket, valamint egy új nyelv támogatásakor is csak ezt az egy erőforrást kell lefordítani
- néhány esetben elkerülhetetlen az OCP elv megsértése, például típusellenőrzés szükséges.
    - Ilyenkor ezt a típusellenőrzést csak egyetlen, jól megválasztott helyen szabad megtenni a kódban

Tell, Don't Ask (mondj, ne kérdezz)

- TDA: *"Ne ellenőrizzük a hívott objektum típusát vagy belső állapotát, mielőtt meghívjuk annak egy metódusát!"*
- egyszerűen csak hívja meg a kliens a szerver metódusát, a szerver pedig a saját belső állapota alapján viselkedjen úgy, ahogy neki kell
- ne toljuk át a felelősségeket a kliensekbe, mert akkor az a DRY elv megsértéséhez fog vezetni
- a TDA elv megsértése azt jelzi, hogy tervezéskor rosszul osztottuk ki a felelősségeket az osztályok között

Law of Demeter (Demeter törvénye)

- Hosszú hívási lánc problémája: a hívó objektum a lánc nagyon távoli elemeitől is függésbe kerül, így, ha azok az elemek változnak, az a hívóra is hatással lehet.
- LoD: *"Ne beszélgess idegenekkel!"*
- Egy objektumnak ismerősnek számít saját maga, a paraméterként kapott objektumok, az attribútumokban tárolt objektumok és azok az objektumok, amelyeket ő maga hoz létre.
- Mindenki más idegennek számít, rajtuk nem szabad függvényhívást végezni.
- megoldás:
    - mindig delegáljuk a hívási lánc maradék részét a következő objektumhoz. Ennek köszönhetően a lánc minden tagja csak a lánc következő objektumával lép kapcsolatba
    - ha a törvény hatására nagyon megugrik a delegáló függvények száma, akkor vizsgáljuk meg, hogy biztosan jól osztottuk-e ki a függőségeket. Ha biztosak vagyunk benne, hogy mindent jól csináltunk, akkor inkább sértsük meg a Demeter törvényt, mint hogy sok delegáló függvényt kelljen karban tartani

Common Closure Principle (együtt változó osztályok elve)

- CCP: *"az együtt változó osztályok ugyanabba a csomagba kerüljenek"*
- így a változás lokalizált
- magas mértékű kohéziót segíti elő

Common Reuse Principle (közös újrahasznosítás elve)

- CRP: *"a nem együtt használt osztályok külön csomagba kerüljenek"*
- ne kötelezzük a klienseket olyan csomagoktól való függésre, amelyeket nem használnak
- ez egyben az ISP alkalmazása csomagokra
- magas mértékű kohéziót segíti elő

Release Reause Equivalency Principle (kiadott komponensek újrahasznosításának elve)

- REP: *"általuk kiadott szoftver komponenseket akkor fognak csak mások használni, ha azokhoz megfelelő támogatást és verziózást biztosítunk"*
- különben a komponensek hosszútávon nem megbízhatóak

Objektumorientált tervezési heurisztikák

Osztályok tervezésével kapcsolatos heurisztikák

1. "Az attribútumok mindig legyenek privátok!"
2. "Ne használjuk másik osztály nempublikus tagjait!"
3. "Minimalizáljuk a publikus metódusok számát!"
4. "Implementáljuk a sztenderd metódusokat"
5. "Egy osztály ne függjön az őt használó osztályoktól, beleértve a leszármazottjait is!"
6. "Egy osztály csak egy absztrakcióval rendelkezzen!"
7. "Az összetartozó adatot és viselkedést tartsuk egy helyen, tehát egy osztályban!"
8. "A metódusok használjanak minél több attribútumot és metódust a saját osztályukból"
9. "A viselkedést modellezük, ne a szerepeket!"

"Az attribútumok mindig legyenek privátok!"

- publikus attribútumok problémái
    - a kliens osztályok belelátnak a szerver osztály belső reprezentációjába
        - ha elkezdenek ehhez kötődni, akkor megnő a csatolás a kliens és szerver osztály között
        - ha szerver belső reprezentációja később megváltozik, akkor a kliens osztályokat is át kell írni
    - egyes attribútumok (pl: hónap, nap) értékei csak bizonyos intervallumokon belül érvényesek
        - amennyiben egy kliens osztály direktben állítja ezeknek az értékét, neki kell figyelnie a konzisztenciára
        - ez kódduplikációhoz vezet, és túlságosan nagy felelősséget hagy a kliensekre
- Publikus és protected attribútumok helyett használjuk privát attribútumokat, a hozzáférést pedig publikus és protected láthatóságú függvényeken keresztül korlátozzuk, amelyek tudnak figyelni az objektum belső állapotának konzisztenciájára
    - ezáltal a konzisztenciát egy helyen, a szerverben biztosítjuk, nem pedig sok helyen megismételve, a kliensekben
- ha a szabályt betartjuk, az osztály belső reprezentációjának megváltozása nem lesz kihatással a kliensekre
- a szabály alól kivételt képeznek a statikus konstans attribútumok, mert ezeknek az értékét a kliensek nem tudják befolyásolni

"Ne használjuk másik osztály nempublikus tagjait!"

- más osztály nem publikus adattagjaihoz való hozzáférés problémája
    - pl.: a sorosító osztály a pont osztály belső reprezentációját használja
    - a sorosító osztály túlságosan erősen függ, erősen csatolódik a pont osztályhoz
    - ha a pont osztály belső reprezentációja megváltozik, a sorosító osztály is át kell írni

- ha esetleg egy másik fejlesztő megszegte az előző szabályt, és a szerver osztályban nem csak privát attribútumokat használt, akkor ne engedjünk a csábításnak, és a kliens osztályunkban ne építsünk azokra az attribútumokra
- ha egy másik osztályból nem csak publikus tagokat használunk, az jelentheti azt, hogy a két osztályt össze kellene vonni egy közös osztályba
    - így az erős csatolást egy erős kohézióra tudjuk lecserélni
    - természetesen ehhez az kell, hogy az összevont osztály minden tagja erős kohézióban legyen egymással
    - ha ez nem teljesül, akkor daraboljuk fel máshogy az osztályokat
- a szabály alól kivételt képeznek azok az esetek amikor valamilyen könyvtárat, vagy keretrendszert készítünk, és ennek bizonyos részeit szeretnénk elrejteni a külvilág elől
- kivételt képez a tesztelés is, ahol a tesz indításához az objektum belső állapotát valamilyen speciális állapotba kell hozni
- a megoldás tehát az, hogy csak publikus metódusokon vagy propertyken keresztül férjünk hozzá egy másik osztályhoz

"Minimalizáljuk a publikus metódusok számát!"

- túl sok publikus metódus problémái
    - egy kliens osztály tipikusan kevés metódust hív a másik osztályból
    - nagyon nehéz megtalálni azt a metódust, amit éppen keresünk
    - egy dolgot akár többféleképpen is meg lehet csinálni
    - ha túl sok protected és private függvényt publikussá teszünk, az az objektum belső állapotáról is elárulhat titkokat - ez sérti az információrejtés elvét
- ne publikáljunk feleslegesen privát és protected függvényeket
- ne szemeteljük feleslegesen tele egy osztály publikus interfészét
- a kliensek csak azokat a függvényeket lássák, amikre ténylegesen szükségük van
- ez az ISP egy gyakorlati megvalósítása
- figyeljünk arra is, hogy a publikus függvényeken keresztül egy dolgot lehetőleg csak egyféleképpen lehessen megcsinálni
- így minden fejlesztő egyetlen, konzisztens módon fogja használni az osztályt

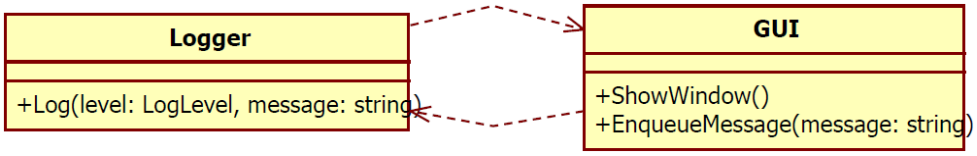
"Implementáljuk a sztenderd metódusokat"

- ilyenek: sztringgé alakítás, összehasonlítás és a hash kód generálás
- ha ezek működnek, az nagyon megkönnyítheti a fejlesztők életét, sőt, még a tesztelésnél is nagyon hasznosak tudnak lenni
- sztenderd metódusok:
    - C#: ToString(), Equals(), GetHashCode()
    - Java: toString(), equals(), hashCode()
    - C++: másolókonstruktor, operator=, operator==, operator<< to an ostream

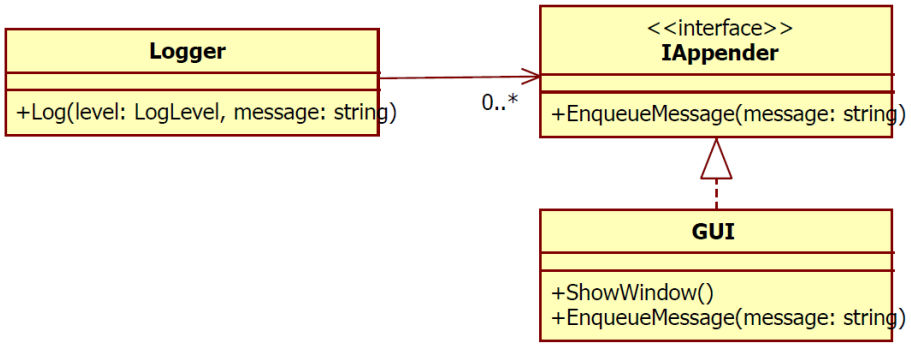
"Egy osztály ne függjön az őt használó osztályoktól, beleértve a leszármazottjait is!"

- körkörös függőség problémája
    - a körkörös függésben lévő osztályok önállóan nem használhatóak újra, csak a többiekkel együtt
    - leszármazás esetén a leszármazott már alapból függ az őstől, így ellenjavallt, hogy az ős függjön a leszármazottjaitól

- ha egy osztály függ a leszármazottjaitól, akkor elkerülhetetlen, hogy a később bevezetett leszármazottaktól is függjön majd
- ez sérti az OCP elvet, és nem is mindig megvalósítható, mert lehet, hogy a leszármazottak egy másik könyvtárban vannak implementálva, és nincsen már hozzáférésünk az ősnek a könyvtárához
- a körkörös függőség sérti az ADP elvet is

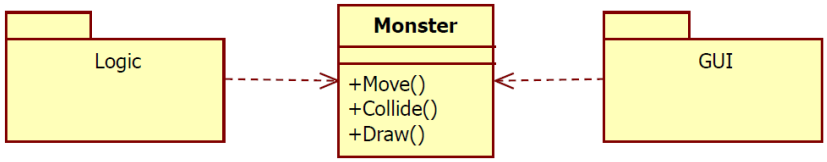


- ha mégis ilyesmire találnánk példát, próbáljuk meg csökkenteni az osztályok közötti függőséget az ISP segítségével vagy oldjuk fel teljesen DIP segítségével
- ellenőrizzük azt is, hogy a felelősségek jól vannak-e kiosztva, vagy esetleg át kell őket rendezni a két osztály között
- esetleg fontoljuk meg a két osztály összevonását, ha a kohézió megfelelően erős
- mindenképpen figyeljünk arra, hogy egy osztály ne függjön a leszármazottjaitól
- a felvetett probléma megoldható például az ISP és a DIP segítségével
  - a GUI EnqueMessage() metódusát kiemeljük egy IAppender interfészbe
  - a Logger csak ettől függ, a GUI pedig ezt implementálja
  - így megszűnik a körkörös függőség a GUI és a Logger között



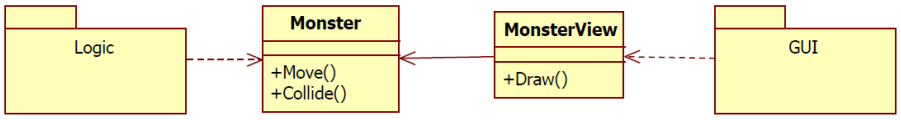
"Egy osztály csak egy absztrakcióval rendelkezzen!"

- túl sok felelősség problémája
  - túl sok absztrakciós szint keveredik az osztályban
  - példák absztrakciós szintekre: grafika, üzleti logika, adatbázis kezelés, hálózatkezelés, naplózás
  - ha ezek közül több is keveredik egy osztályban, akkor az osztálynak több oka is lehet a változásra, sérül az SRP

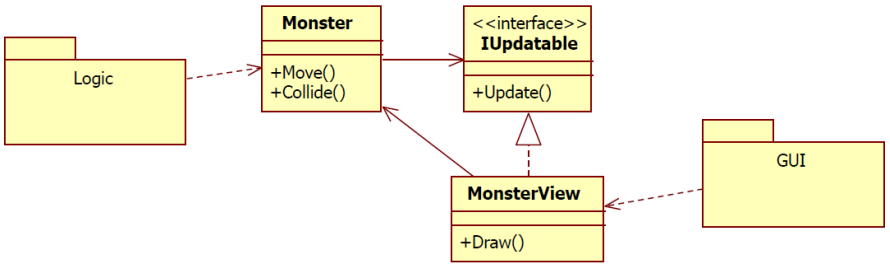


- ha egy osztályban keverednek az absztrakciós szintek, akkor gondolkodjunk el rajta, hogy hogyan lehetne egy osztályt feldarabolni több kisebb osztályra, vagy legalább az ISP segítségével enyhíteni a problémán
- az absztrakciós szintek keveredésekhez vezethet az is, ha felelősségek rossz helyre vannak allokálva (pl.: a kliens osztályok ellenőriznek olyan feltételeket, melyek ellenőrzése a szerver osztály feladata lenne)

- kivételt képez ez alól, ha a felelősség úgy van kiosztva, hogy a DRY elv nem sérül
    - ilyenek például a Visitor és Strategy tervezési minták
- a felvetett probléma megoldásai az alábbiak lehetnek
    - például a szörny osztályt kettévághatjuk az absztrakciók mentén két különböző osztállyá

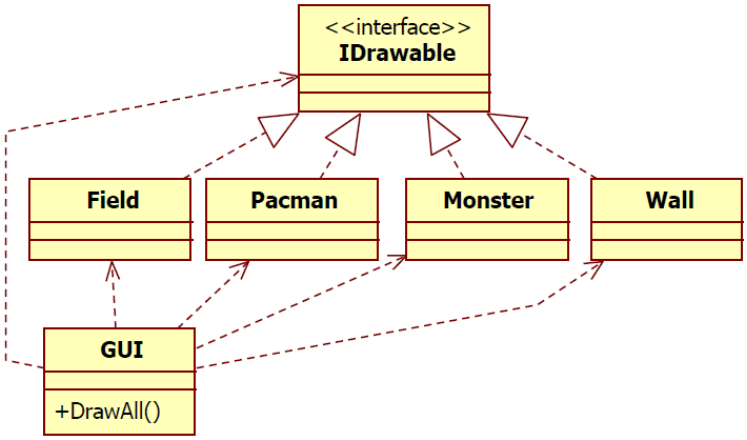


    - egy másik lehetőség a frissíthetőséget, mint absztrakciót külön kiemelni egy interfésszé, amit a grafika implementál, a logikai rész pedig csak ettől az interfésztől függ



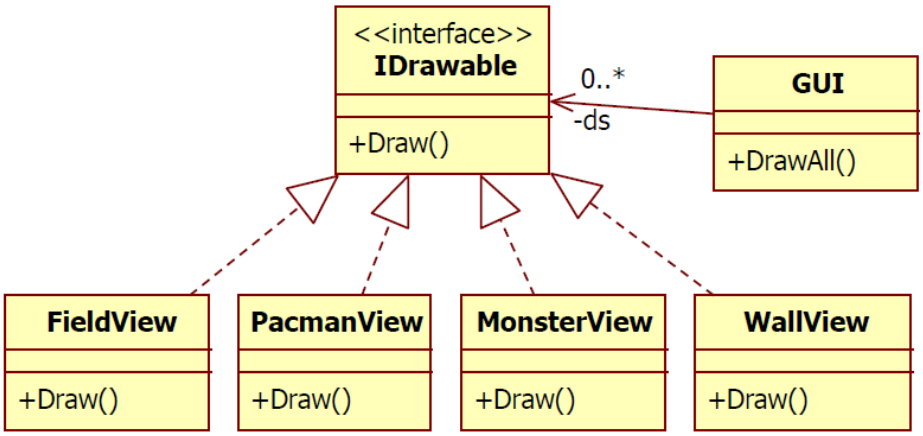
"Az összetartozó adatot és viselkedést tartsuk egy helyen, tehát egy osztályban!"

- a viselkedés az adat különválasztásának problémája
    - az alábbi példában a GUI DrawAll() függvénye az összes lehetséges kirajzolandó osztályt ismeri
        - itt sérül az OCP elv, hiszen ha bejön egy új szereplő a játékba, át kell írnunk a DrawAll() függvényt
        - az is problémát okoz, hogy a rajzolás, mint viselkedés külön van választva az adattól, ami alapján a rajzolást el kell végezni



    - gyakran előfordul az is, hogy az adatot és a viselkedést tartalmazó osztályok körkörös függőségbe kerülnek egymással
        - ennek akár rendszerszintű hatása is lehet, ahol a rendszer egyes komponensei körkörös függőségbe kerülnek egymással
        - így az egyes komponensek önállóan nem lesznek újrahasznosíthatók
    - a problémát okozhatja a rossz tervezés is
        - tervezésnél mindig a viselkedést kell szétosztani, az adatok csak támogató jellegűek, nem ők az elsődleges tervezési szempont

        - ha ezt elrontjuk, és nem objektumorientáltan, hanem procedurálisan tervezünk, akkor az eredmény tipikusan egy csomó adattároló osztály, és az ezeken operáló adatmanipulátor osztályok
        - ennek következménye a TDA és a DRY elvek megsértése
- ha körkörös függőséget találunk két osztály között, gondolkodjunk el azon, hogy össze lehet-e vonni őket egy osztállyá
- figyeljünk a TDA és a DRY elvek megsértésére, és a kliensekben implementált ismétlődő funkcionalitást vigyük át a szerverbe
- mindig legyen gyanús az, ha adattároló, és azokon manipuláló osztályokat találunk
    - ilyenkor a műveleteket inkább rendeljük az adattároló osztályokhoz, és ezáltal viselkedéssel ruházzuk fel őket
- a szabály alól kivételt képeznek azok az osztályok, melyek kizárólag adatcseréle valók
    - ilyenek: adatbázisleképzést megvalósító osztályok, hálózati kommunikációt támogató osztályok
- kivételt képeznek azok az osztályok is, amelyek megfontolt tervezői döntés alapján kiszervezett működést tartalmaznak
    - ilyenek: Strategy és Visitor tervezési minta
- a példánk megoldása tehát az, hogy minden szereplő egy saját nézet osztállyal rendelkezik, mely tartalmazza a rajzoláshoz szükséges információkat, mint például a pozíció, színek stb…
    - a nézet osztályok pedig saját maguk végzik a rajzolást a bennük tárolt információk alapján
    - így a rajzoláshoz szükséges információk és a rajzolás, mint viselkedés egy helyen vannak



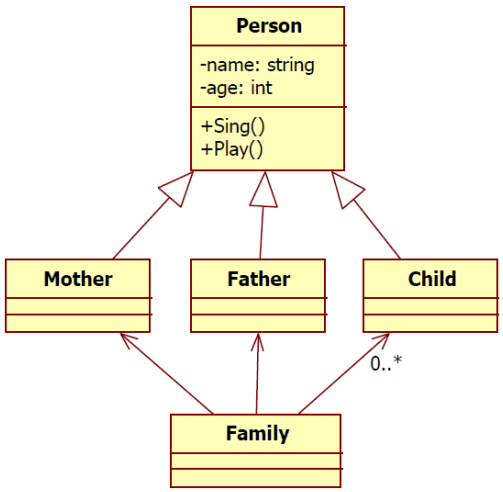
"A metódusok használjanak minél több attribútumot és metódust a saját osztályukból"

- túl sok független felelősség példája
    - nincs meg a kohézió az osztályon belül
    - sérül a SRP
    - felmerül a gyanú, hogy az osztály önmagában egy "isten-osztály": mindent ő irányít
- ne keverjünk egy osztályban olyan adatokat és viselkedéseket, amelyek nem tartoznak össze
- kerüljük az isten-osztályokat, amelyek mindent irányítani próbálnak, és ezáltal sok független viselkedés keveredik bennük

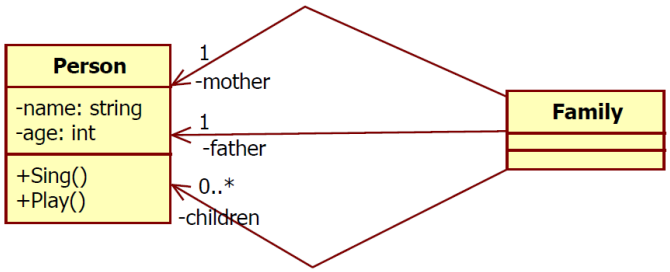
    - ha mégis ilyet találunk, akkor daraboljuk fel a felelősségei mentén
- a szabály alól kivételt képeznek az adatbázis-leképzést megvalósító osztályok és a hálózati kommunikációt támogató osztályok
- ugyancsak kivételt képeznek az Utility osztályok, melyek mindenki számára hasznos, de senkihez sem hozzárendelhető funkciókat tartalmaznak
    - ilyenek például: matematikai függvények (sinus, cosinus, négyzetgyök)

"A viselkedést modellezük, ne a szerepeket!"

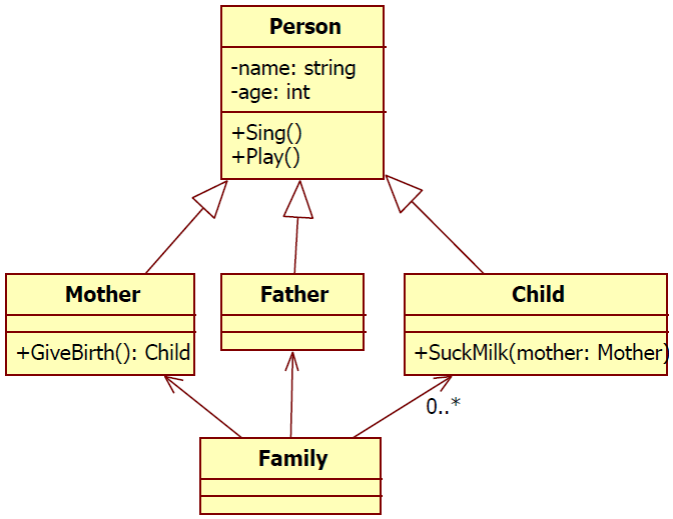
- a leszármazottak nem adnak hozzá semmit sem az ősosztály viselkedéséhez
    - a leszármazottak lehetnének az ős különböző példányai is, mivel nem adnak hozzá új működést
    - ilyenkor a leszármazottak viselkedése megegyezik az ősével, felesleges őket külön típusként modellezni
    - az is hibás viselkedés, ha bevezet ugyan egy újabb attribútumot a levél, de nem tartozik hozzá külön viselkedés, ilyenkor ugyanis nincs, aki használja az attribútumot
    - tanulság: újabb vagy felülírt viselkedés nélkül nincs sok értelme a leszármazottnak



- újabb vagy felülírt viselkedés nélkül nincs sok értelme a leszármazottnak
- ne különálló objektumokat modellezzünk, hanem az ő közös viselkedésüket leíró osztályokat
- természetesen, ha egy ős absztrakt osztály vagy interfész, akkor az UML diagramon lehet üres a leszármazott osztály, de implementációs szinten implementálni fogja az ős metódusait
- érdemes azt is megvizsgálni, hogy az osztály publikus interfészének mely része melyik szerepkörben érvényes
    - ha van olyan függvény, amely csak valamely szerepkörben érvényes, akkor ahhoz a szerepkörhöz külön osztályt kell felvenni, és ezt a függvényt ahhoz az osztályhoz rendelni
    - ha az osztály publikus interfészében van olyan függvény, amelyet valamely szerepkör nem használ, bár használhatná, akkor az még ugyanaz az osztály, nem kell külön szerepkörként modellezni
- az öröklési hierarchiában szerepelhetnek üres levelek (legalsó leszármazottak), amennyiben a viselkedésük külön helyre van kiszervezve
    - ehhez hasonló a Visitor tervezési minta, de ott is legalább a Visitor meghívását implementálni kell a levélben
- megoldás: ha a leszármazottak nem adnak hozzá viselkedést az ősosztályhoz, akkor ne az osztály leszármazottjaiként, hanem annak példányaiként modellezzük őket



- ha mégis vannak olyan funkciók, amelyek csak adott szerepkörben érvényesek, akkor modellezük csak ezeket önálló osztályként



Felelősségek kiosztásával kapcsolatos heurisztikák

1. "A felelősségeket egyenletesen osszuk szét!"
2. "Kerüljük az isten-osztályokat"
3. "Kerüljük azokat az osztályokat, amelyeknek függvényeknek kellene lenniük!"
4. "Modellezzük a valódi világ működését"
5. "Modellezzünk a megfelelő absztrakciós szinten!"
6. "Modellezésnél maradjunk a rendszer határain beül!"
7. "Mindig a nézet függjön a modelltől, sosem a modell a nézettől!"

"A felelősségeket egyenletesen osszuk szét!"

- Ha ezt nem tartjuk be, lesznek olyan osztályok, melyek túl sok felelősséggel rendelkeznek, és lesznek olyanok, melyek túl kevéssel
- ennek egy extrém változat az, amikor egy isten-osztály irányít adattároló osztályokat
- ilyen eset akkor fordul elő, ha felelősségalapú objektumorientált tervezés helyett procedurális tervezést alkalmazunk
    - tehát tervezéskor keressük meg az entitásokat, rendeljünk hozzájuk felelősségeket, és az adatokat csak legvégül adjuk hozzá
- vigyázzunk arra is, hogy ne terheljünk túl egy-egy osztályt túl sok felelősséggel
    - ha ilyen helyzetben találnánk magunkat, rendezzük át a felelősségeket, vagy daraboljuk fel az osztályt

"Kerüljük az isten-osztályokat"

- isten-osztály problémája
    - az osztályok nagy része csak adatokat tárol, és az ezekhez tartozó viselkedés nem náluk van
    - az isten-osztály irányít mindenkit
- ha mégis isten-osztállyal találkoznánk, daraboljuk fel az osztályt a felelősségek mentén, és osszuk fel a felelősségeit a többi osztály között

"Kerüljük a csak adattárolásra használt osztályokat!"

- mivel ezek önálló felelősséggel nem rendelkeznek, könnyen abba hibába eshetünk, hogy más osztályok - tipikusan a TDA megsértése révén - ezeket az osztályokat fogják manipulálni
- ilyenkor célszerű azon elgondolkozni, hogy a manipuláló osztályokban implementált viselkedést inkább az adattároló osztályokhoz kellene rendelni.
- ez alól kivételt képeznek az adatbázis-leképzést és a hálózati üzeneteket reprezentáló osztályok

"Kerüljük azokat az osztályokat, amelyeknek függvényeknek kellene lenniük!"

- ne legyen külön helyen a viselkedés és az adat, mert így az osztályok kohéziója felbomlott
- legyen különösen gyanús az, ha egy osztály neve nem főnév, hanem ige, vagy igéből származik
- legyen gyanús, ha egy osztálynak csak egyetlen egy függvénye van
    - célszerű ilyenkor ezt a viselkedést egy már létező osztályhoz hozzárendelni
- a szabály alól kivételt képez az, ha a viselkedést szándékosan szervezzük ki, például a Vistior vagy a Startegy tervezési minta segítségével
    - ilyenkor a viselkedés leválasztása tudatos tervezői döntés eredménye

"Modellezzük a valódi világ működését"

- ez azért előnyös, mert a valódi világ már eleve működik
- könnyebb megtalálni a szereplőket, könnyebb kitalálni azt, hogy melyik felelősséget kihez rendeljük hozzá
- a modellezésnél célszerű figyelembe venni a fizika törvényeit, többek között a lokalitást
    - a lokalitás azt jelenti, hogy mindenki csak a körülötte lévő dolgokat ismeri, csak velük tud interakcióba lépni
    - például, amikor egy vonatot modellezünk, az első kocsi húzza a másodikat, a második a harmadikat és így tovább
    - mindenki csak az előtte és a mögötte lévő kocsit ismeri, nincs olyan szereplő, a világban, aki az összes kocsit egyszerre lépteti
    - fontos tehát, hogy ne legyen egy globális ügynök, aki irányítja a világot, hanem minden interakció helyben, lokálisan történjen meg
- természetesen lehetnek kivételek, amikor nem ragaszkodunk a valódi világ működéséhez
    - például elektronikus levelezésnél nem kell postagalambokat modellezni
- sok esetben érdemes a valódi világ működését követni, azonban, ha van megfelelő indokunk, tudatos tervezői döntéssel el lehet ettől térni

"Modellezzünk a megfelelő absztrakciós szinten!"

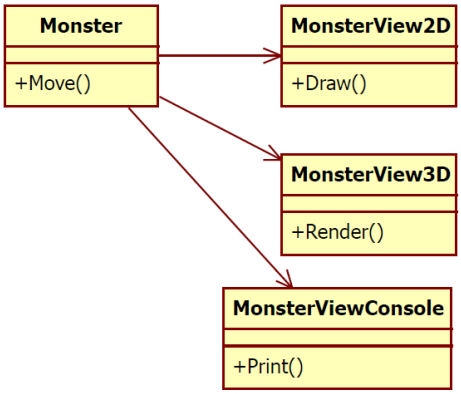
- Nem kell ragaszkodnunk a való világ modellezéséhez, ha az kényelmetlen és nehezen karbantartható programhoz vezet
- vezessünk be nyugodtan ügynököket, hogy az alkalmazás egyes részeit szétcsatoljuk egymástól
    - például az alkalmazás üzleti logikáját a hálózatkezeléstől vagy a grafikától
- tehát nyugodtan vezessünk be absztrakt ügynököket, ha azok hasznos absztrakciót reprezentálnak
    - például bevezethetünk cache-elést a hatékonyság növelésére, habár a való világban nincsen ilyen szereplő
- lehetnek azonban olyan szereplők is a világban, melyek csak túlbonyolítják a terveket, ilyenkor ezek az ügynökök elhagyhatók a modellből
    - például a vonat mozgásának modellezéséhez nem feltétlenül szükséges az erő-ellenerő törvénye
- tehát ez a szabály azt mondja ki, hogy nyugodtan vegyünk fel olyan ügynököket, melyek megkönnyítik a modellezést, és eliminálják azokat az osztályokat, melyeknek nincs értelmes hozzáadott értéke

"Modellezésnél maradjunk a rendszer határain beül!"

- Ehhez természetesen ismerni kell a rendszer határait
- célszerű már a követelmények meghatározásánál ezt a határt megtalálni
- ha nem tiszták a rendszer határai, akkor sok energiánk el fog menni felesleges dolgokra
- például egy pénzkiadó automata határai: képernyő, gombok, kiadó rés
    - a felhasználó, aki kezeli az automatát, nem része az automatának
    - az automata modellezésénél a felhasználót, mint embert nem kell modellezni
- mindig törekedjünk arra, hogy egyértelmű legyen, hogy hol vannak a rendszer határai
- a felhasználók és külső rendszerek felé mindig valamilyen interfészen keresztül kapcsolódunk
- a rendszer határait, és külső interfészeit már a követelményeknél meg kell határozni

"Mindig a nézet függjön a modelltől, sosem a modell a nézettől!"

- probléma: a modell a nézettől függ
    - a példán a Pacman játék 2D-ben, 3D-ben vagy akár konzolon is játszható
    - a logikai működést megvalósító szörny osztály ismeri mind a háromfajta nézetet
    - újabb interfészek bevezetése hatással lesz a modellre
    - például, ha újfajta megjelenítést szeretnénk támogatni, vagy hálózati kommunikációt szeretnénk támogatni
    - a nézet minden változása kihatással van a modellre

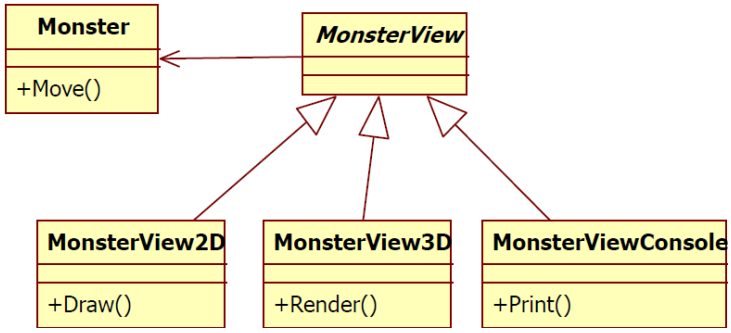


- Ha a nézet függ a modelltől, a modell független lesz a megjelenítéstől és a hálózati kommunikációtól
- amennyiben szükséges, a DIP segítségével meg lehet fordítani a függőség irányát

- kétféle alapvető megoldás létezik:
  - egyik megoldás a pull modell, ahol a grafika folyamatosan lekérdezi a modell állapotát
  - egy másik lehetséges megoldás a push modell, ahol a modell értesíti a grafikát amennyiben valami változott
- mind a push és mind a pull modellben megfigyelhető, hogy függőség iránya a nézettől a modell irányába mutat

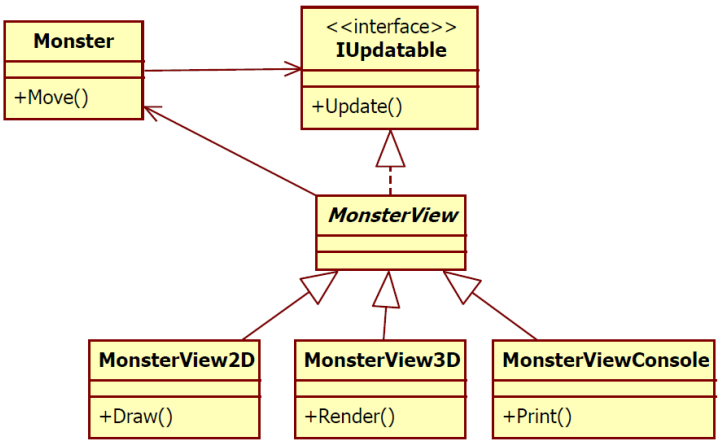
Pull modell

- a grafika folyamatosan lekérdezi a modell állapotát
- ilyenkor a grafikát bizonyos időnként újra rajzoljuk, támaszkodva a modell aktuális állapotára
- természetesen figyelni kell arra, hogy elég gyakran frissítsünk ahhoz, hogy a modell minden változását érzékelni tudjuk
- előfordulhat az is, hogy túlságosan gyakran frissítünk, és így feleslegesen rajzolunk mindig újra és újra, hiszen közben a modell állapota nem változott



Push modell

- a modell értesíti a grafikát amennyiben valami változott
- ilyenkor mindig csak akkor rajzolunk újra, amikor ténylegesen szükséges
- ehhez a megoldáshoz azonban szükséges, hogy a modell valamilyen szinten ismerje a grafikát
- ez az ismeretség azonban jól leválasztható egy interfész segítségével
- ha a modell része az IUpdatable interfész, és a grafika ezt implementálja, akkor a modellnek elegendő csak ezt az interfészt ismernie, a konkrét grafikai megvalósítástól nem függ
- a grafika természetesen továbbra is ismeri a modellt, hiszen a megjelenítéshez szükség van a modell állapotának lekérdezésére

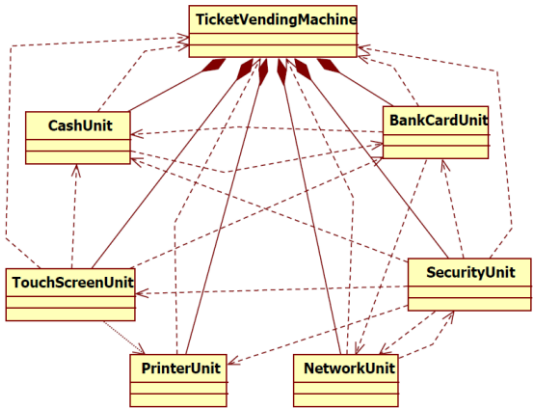


Asszociációkkal kapcsolatos heurisztikák

1. "Minimalizáljuk az együttműködő osztályok számát!"
2. "Minimalizáljuk az együttműködő osztályok között használt metódusok számát!"
3. "Osszuk szét a felelősségeket a tartalmazás mentén!"
4. "Asszociáció helyett preferáljuk a tartalmazást"
5. "A tartalmazó objektum ne csak egyszerűen tartalmazza, hanem használja is a tartalmazott objektumokat!"
6. "A tartalmazott objektum ne használja a tartalmazó objektumot!"
7. "A tartalmazott objektumok ne beszélgessenek egymással közvetlenül!"

Túl sok asszociáció problémája

- túl sok a keresztfüggőség
- úgy néz ki program felépítése, mint egy spagetti: bármelyik komponenst is fogjuk meg, jön vele az összes többi
- nem lehet a komponenseket önállóan újrahasznosítani
- a következő szabályok ezekre a problémákra fognak megoldást adni



"Minimalizáljuk az együttműködő osztályok számát!"

- ha túl sok osztály működik együtt, túl sok közöttük a keresztfüggőség, túl nagy a csatolás
- ha szükséges, az ISP és a DIP segítségével lehet csökkenteni a függőségek számát, vagy megfordítani a függőségek irányát

"Minimalizáljuk az együttműködő osztályok között használt metódusok számát!"

- minél több különböző metódust hívnak egymás között az együttműködő osztályok, annál erősebb közöttük a csatolás
- sokszor ez annak a jele is lehet, hogy megsértjük a TDA és a DRY elveket
- érdemes azon is elgondolkozni, hogy a felelősségek jól vannak-e elosztva
- lehet az is a probléma, hogy isten-osztályt hoztunk létre, amely mindenkit irányítani akar

"Osszuk szét a felelősségeket a tartalmazás mentén!"

- ne a tartalmazó osztály csináljon mindent, hanem a tartalmazott komponensek viselkedéséből jöjjön ki a tartalmazó osztály működése
- osszuk szét a felelősségeket mély és keskeny tartalmazási hierarchiákban
- a tartalmazó osztályok fekete-dobozok legyenek, külső használói ne tudjanak semmit a belső struktúrájáról, vagyis, hogy milyen komponensekből épül fel
- a belső komponenseket ne adjuk ki a külvilág felé, velük csak a tartalmazó komponens kommunikálhat
- ha ezt nem tartjuk be, a Demeter törvényt is megsértjük

"Asszociáció helyett preferáljuk a tartalmazást"

- az asszociáció ugyanis nem fekete-dobozként viselkedik
- az osztály külső használói tudhatnak az asszociációban lévő objektumokról, függhetnek tőlük
- ha tehát lehetőségünk van, preferáljuk a tartalmazást az asszociációval szemben
- vigyázzunk azonban, mert a tartalmazások nem alkothatnak kört, és egy objektumot nem tartalmazhatnak ketten egyszerre
- ha tehát tartalmazást nem tudunk használni nem tehetünk mást, mint hogy maradunk az asszociációnál
- ugyancsak kivétel a szabály alól, ha a belső objektumok kívülről, másoknak is látniuk kell

"A tartalmazó objektum ne csak egyszerűen tartalmazza, hanem használja is a tartalmazott objektumokat!"

- ha ugyanis nem használjuk a tartalmazott objektumokat, akkor valószínűleg feleslegesen tartalmazzuk őket, vagy kiadjuk a külvilág felé, megsértve a Demeter-törvényt
- a tartalmazás tehát egyben felhasználást is implikál
- a Demeter-törvényt betartva, a konténernek kell továbbítania a kiadott hívásokat az objektumok irányába
- a Façade tervezési minta is hasonló elveken alapul
- a szabály alól kivételt képeznek a gyűjtemény osztályok, mint például egy lista vagy halmaz, mert ezek csak tárolják, de nem használják a tartalmazott objektumokat

"A tartalmazott objektum ne használja a tartalmazó objektumot!"

- ez azért fontos, mert a konténer már eleve függ a tartalmazott objektumtól, és nem jó az, ha a másik irányba is bevezetünk egy függőséget
- azért sem jó, ha a tartalmazott objektum függ a konténertől, mert akkor a tartalmazott objektumot nem lehet kiemelni, és önmagában újrahasznosítani
- amennyiben mégis van függőség a tartalmazott objektumtól a konténer felé, akkor a DIP és ISP segítségével ezt meg lehet szűntetni

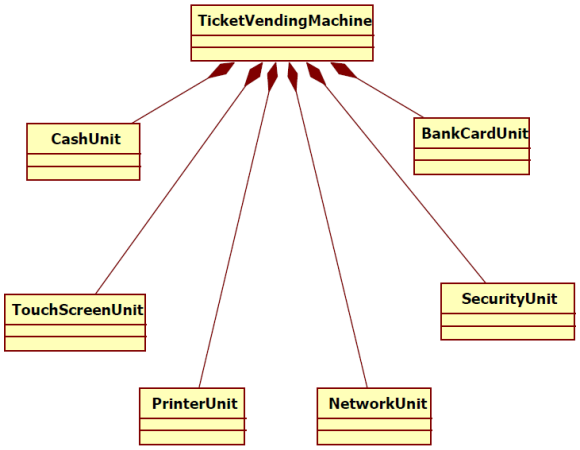
"A tartalmazott objektumok ne beszélgessenek egymással közvetlenül!"

- a konténer már eleve ismeri a tartalmazott objektumokat
- ha a tartalmazott objektumok is ismerik egymást, az már túl sok keresztfüggőséghez fog vezetni, nem lehet majd őket önállóan újrahasznosítani
- a szabály tehát az, hogy a tartalmazott objektumok egymással közvetlenül ne beszélgessenek, a konténeren keresztül oldják meg inkább a kommunikációt
- jobb az, ha a konténer és a komponensek között van függőség, mintha a komponensek között lennének keresztfüggőségek
- n darab komponens esetén előbbi esetben 2n db, utóbbi esetben akár n^2 függőség is lehetséges
- sok esetben elegendő az is, hogy a konténer oldja meg a komponensek közti információcserét anélkül, hogy a komponensek ismernék a konténert
    - így működik például a Mediátor tervezési minta

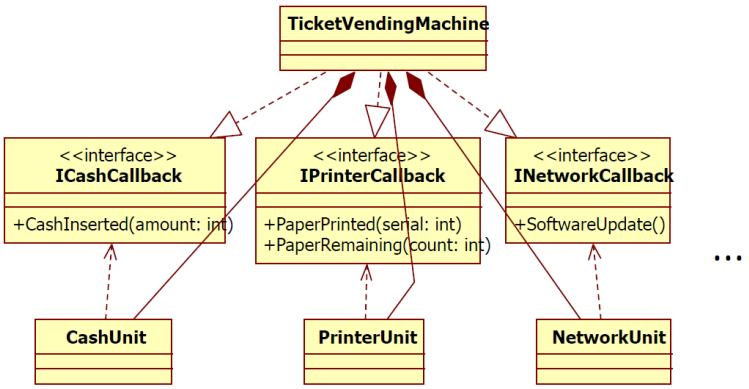
- ha valami miatt a komponensnek mégis csak értesíteni kell tudnia a konténert, például, hogy bedobtak egy érmét az automatába, akkor a DIP vagy az ISP segítségével egy call-back interfészen keresztül ezt meg lehet tenni
    - ezáltal eliminálni lehet a körkörös függőséget a konténer és a tartalmazott objektum között
    - így működik az Observer tervezési minta, amely a C# nyelvben eseményeknek felel meg

Konténerek és komponensek közötti függőségek megoldása

- Ha lehetséges, törekedjünk arra a megoldásra, hogy a konténer tartalmazza a komponenseket, ő oldja meg közöttük az információcserét
    - a komponensek egymást nem ismerik, mégis ők implementálják a felelősségeket
    - a konténer csak egy minimális összekötő és delegáló szerepet lát el



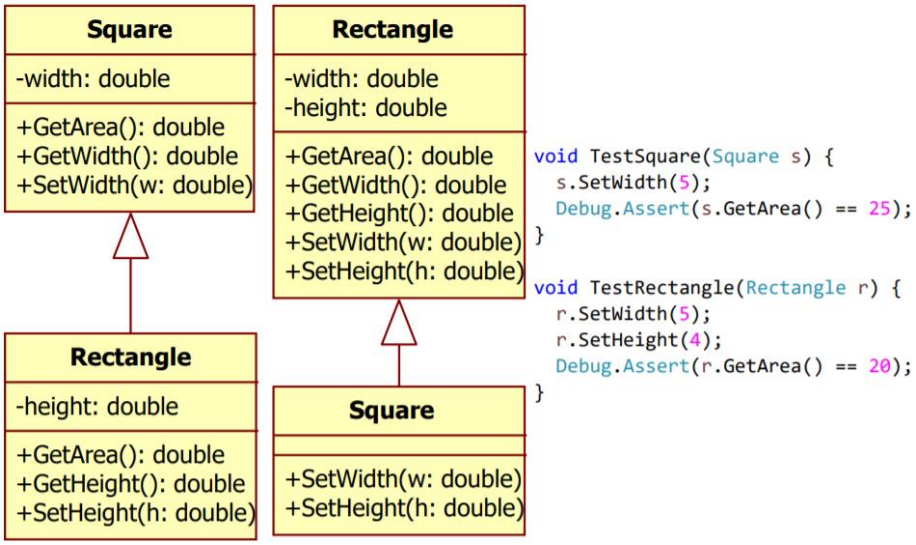
- Ha az előző egyszerű megoldás nem működik, akkor az alábbi megoldást célszerű követni:
    - itt minden egyes komponens definiál magának egy callback interfészt
    - a konténer pedig ezt az interfészt implementálja
    - alapesetben a konténer a hívásokat delegálja a komponensek felé, de azok mégis vissza tudják hívni, és tudják értesíteni a konténert, ha valami történt
    - fontos megjegyezni, hogy itt a komponenst alkotó modul a komponensből és az ő callback interfészéből áll
    - ha így tekintjük, a komponensek moduljai között nincsenek keresztfüggőségek, és a komponensek is csak közvetve, egy interfészen keresztül függnek a konténertől, nincsen közvetlen függőség
    - a komponenseket alkotó modulok tehát önmagukban újrahasznosíthatóak



Öröklődéssel kapcsolatos heurisztikák

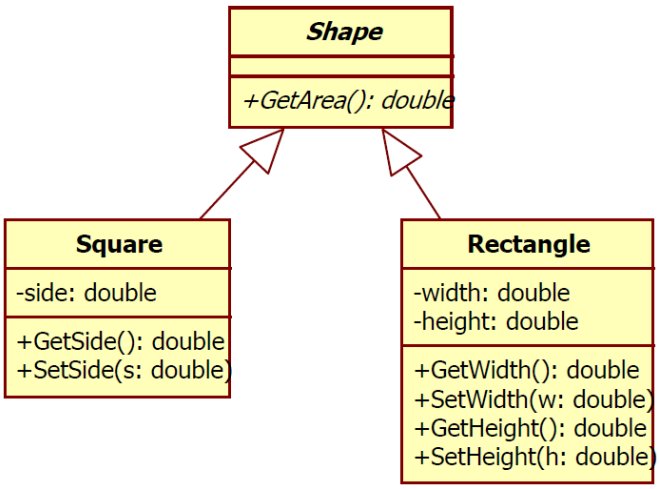
"Az öröklődés célja mindig a viselkedés újrahasznosítása!"

- a négyzet-téglalap probléma
  - ha írunk teszteket a négyzetre és a téglalapra is, ezek a tesztek megmutatják, hogy mi az a viselkedés, amit a kliens elvár
  - például, ha egy négyzet oldalát 5-re állítjuk, akkor azt várjuk, hogy a területe 25 lesz
  - ha egy téglalap oldalait 5 és 4 értékre állítjuk, akkor azt várjuk, hogy a területe 20 lesz
  - ha a téglalap a négyzet leszármazottja, és az első tesztesetnek egy 3 szélességű, négy magasságú téglalapot adunk át paraméterül, majd a szélességet 5-re állítjuk, a területe 20 lesz, és nem a négyzettől elvárt 25
  - ha a négyzet a téglalap leszármazottja, és a második tesztesetnek egy tetszőleges méretű négyzetet adunk át paraméterül, majd az oldalhosszúságát először 5-re, majd 4-re állítjuk, akkor a területe 16-lesz a téglalaptól elvárt 20 helyett



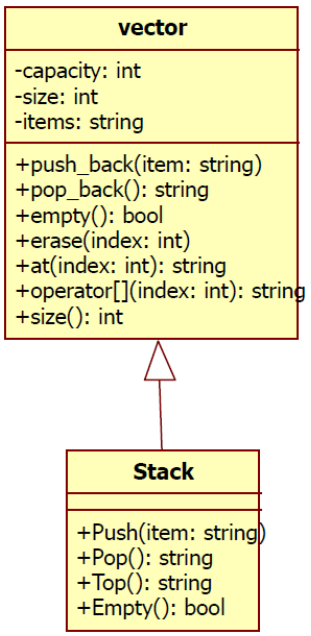
  - a négyzet-téglalap probléma alapját kétféle szemléletmód adja
  - az első szemléletmód azt mondja, hogy a négyzetnek már úgyis van egy oldalhossza, azt újra lehet hasznosítani a téglalapban
  - ez a szemléletmód azért hibás, mert már korábban is említettük, hogy az öröklődés sosem az adatok újrahasznosítását jelenti, hanem a viselkedés újrahasznosítását
  - márpedig a felírt tesztek alapján egy téglalap egyáltalán nem úgy viselkedik, mint egy négyzet
  - a második szemléletmód a matematikából származik, vagyis, hogy a négyzet egy speciális téglalap, amelynek mindkét oldala egyenlő hosszúságú
  - viselkedés szempontjából azonban itt sem helyes a gondolkodásmód
- a Liskov elv alapján a leszármazott nem léphet ki azokból a keretekből, melyet az ős biztosít
- az öröklődést tilos adatok újrahasznosítására felhasználni
- adatok újrahasználására a tartalmazás és a delegáció való
- a négyzet-téglalap probléma megoldása:

- a Liskov-féle helyettesítési elv betartásával az alábbi megoldás vezethet eredményre, ahol a négyzetnek és a téglalapnak van egy közös alakzat őse, amely definiálja közös viselkedést



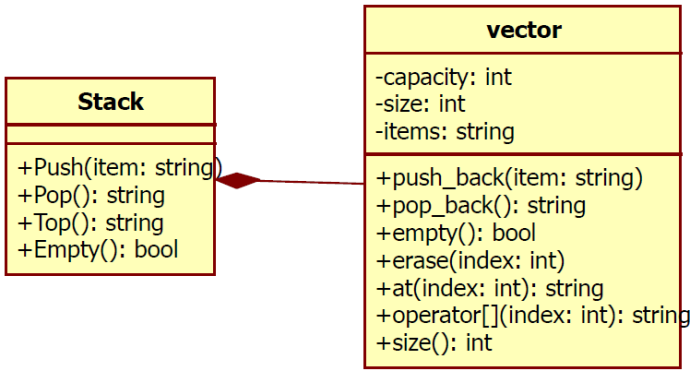
"Az öröklés helyett preferáljuk a tartalmazást!"

- Meglévő tárolóból való származtatás problémája
  - c++ban előfordul, hogy valaki egy kollekciót egy másik kollekcióból származtat
    - például egy verem implementációját a standard vectorból
    - az érvelés emögött az, hogy a vectornak sok olyan hasznos művelete lesz, amely a veremnél is releváns lesz
    - a gondot az okozza, hogy a vectornak nagyon sok olyan művelete van, amely a veremnél egyáltalán nem releváns
    - így a vermet használó kliensek olyan függvényektől is függnek, melyeket egyáltalán nem használnak
    - ez az ISP megsértése



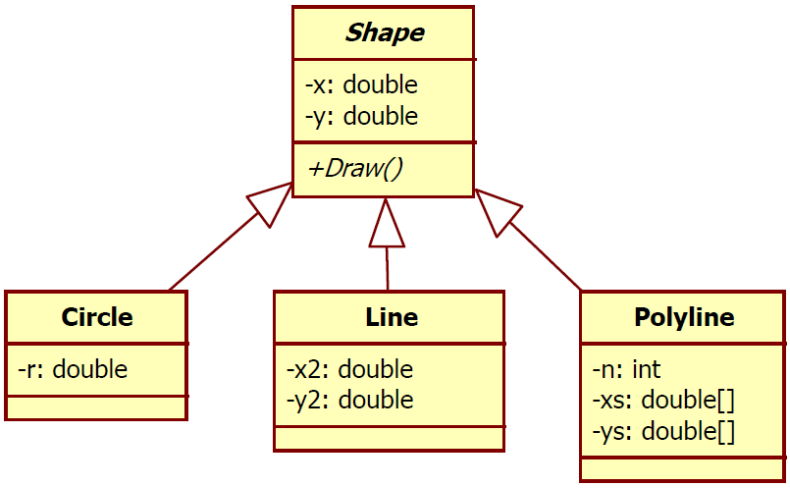
  - mondhatná valaki, hogy C++ban a privát öröklés megoldás lenne erre, de ez nem jelent mentséget, mivel az ősön keresztül továbbra is elérhetőek lehetnek a műveletek
    - privát örökléssel a verem osztály megsértené a Liskov elvet, hiszen a verem osztályt nem lehetne vektorként használni
    - ha viszont nem privát öröklést használunk, akkor a vektor publikus függvényei kiszivárognak a verem publikus interfészére
    - ezáltal a verem túl sok belső implementációs részletet elárul
    - olyan műveletek is elérhető válnak, melyek egy verem esetén nem használhatók (például egy elem beszúrás a verem közepébe)
  - az öröklés tehát nem egy feketedoboz jellegű újrahasznosítás, a belső implementációt sokkal nehezebb lesz később megváltoztatni
- ha nem vagyunk magunkban biztosak, akkor válasszuk a tartalmazást az öröklődés helyett

- adatok újrahasznosítására mindig a tartalmazás való
- csak akkor használjunk öröklődést, ha a viselkedést is újrahasznosítjuk, illetve nem lépünk ki az ősnek a kereteiből, nem sértjük meg a Liskov-elvet
    - ne sértsük meg az ősnek a szerződését, vizsgáljuk meg elő- és utófeltételeket, illetve az invariánsokat!
- viselkedést nem csak örökléssel lehet hozzáadni egy osztályhoz, hanem delegálással is, ezt csinálja a Dekorátor tervezési minta
    - így lehet például Javaban szálbiztossá tenni egy kollekciót, hogy becsomagoljuk egy szálbiztos dekorátorba
- visszatérve a példánkra, a megoldás az, hogy a verem tartalmaz egy vektort, és a verem műveletei ehhez vannak delegálva
    - a vektor kívülről nem érhető el, csak a Push(), Pop(), Top() és Empty() műveletek alkotják a verem publikus interfészét



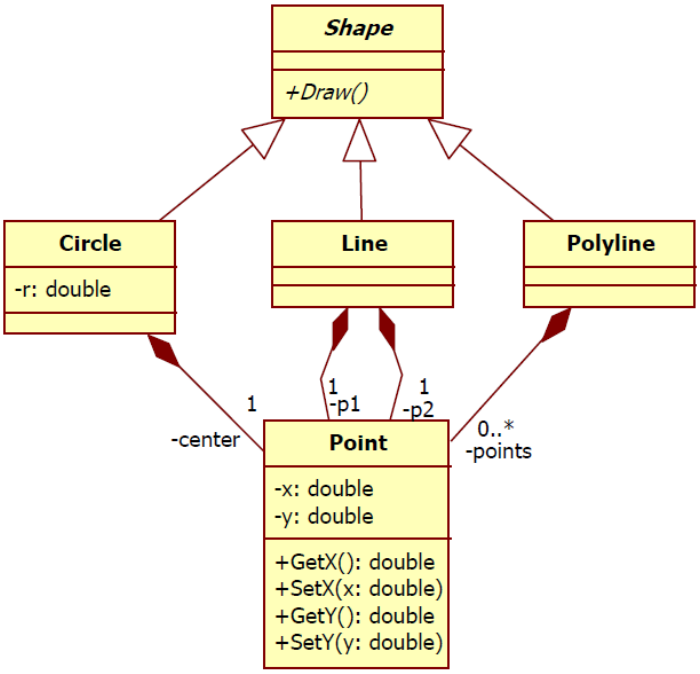
"A közös viselkedéssel nem rendelkező közös adat tartalmazás relációban legyen!"

- probléma: az ábrán a fő gond az, hogy a Polyline pontjainak koordinátái az xs és ys tömbökben vannak tárolva
    - a kérdés az, hogy mit kezdjünk az alakzatból megörökölt x és y koordinátákkal
    - a gond az, hogy az alakzatban az x és y értékekhez nem tartozik olyan közös viselkedés, amelyet a leszármazottak megörökölhetnének
    - további problémát okozhat az is, hogy az xs és ys párhuzamos tömbök, és ha valamilyen implementációs hibát vétünk, a bennük tárolt értékpárok szétcsúszhatnak egymástól



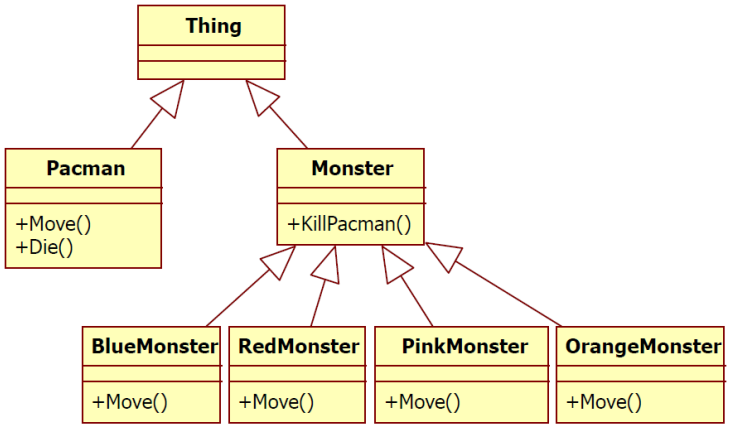
- közös adatot csak akkor vigyünk fel az ősbe, ha közös viselkedés is tartozik hozzájuk

- a probléma helyes megoldása tehát az, hogy az x,y koordinátákat egy külön pont osztályba kiszervezzük, és minden alakzat csak a számára releváns koordinátákat tárolja
    - így nem örököl meg senki sem felesleges adatot, és a Polyline pontjai is egységesen kezelhetők



"A közös viselkedéssel rendelkező közös adat az ősosztályban legyen definiálva"

- probléma: a szörnyekben a KillPacman implementációja ugyanaz
    - így sérül a DRY elv
- a közös viselkedés egyben közös absztrakciót is jelent
- ha két osztály közös adattal, és ehhez tartozó közös viselkedéssel rendelkezik, akkor ennek a két osztálynak egy közös ősből kellene származnia, amely tartalmazza a közös adatot és a hozzá tartozó közös viselkedést
- a szabály alól kivétel lehet az, ha többszörös öröklésre van szükségünk, de a programnyelv ezt nem támogatja
    - ilyenkor az ismétlődés elkerülésére delegáció használható, érdemes ilyenkor a Strategy tervezési mintát alkalmazni
- a megoldás az lehet, ha a KilPacman() függvényt egy közös ősbe helyezzük el, ám ez nem lehet a Thing osztály, mert a Pacman is abból származik
    - tehát a szörnyeknek készítenünk kell egy közös őst, és ebben kell elhelyezni a KillPacman() függvényt
    - a KillPacman() függvény azért nem vihető magasabbra, mert a Pacman számára ennek a viselkedésnek nincs értelme

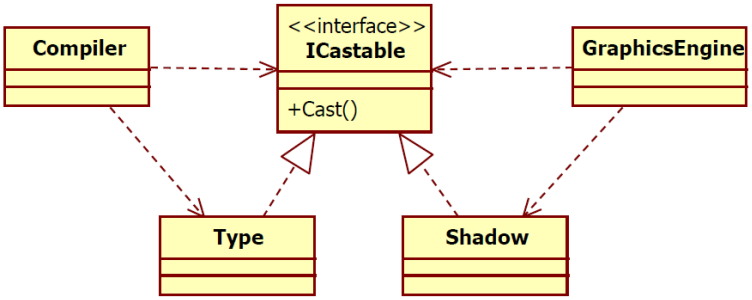


"A közös viselkedés és közös adat minél magasabban legyen az öröklési hierarchiában!"

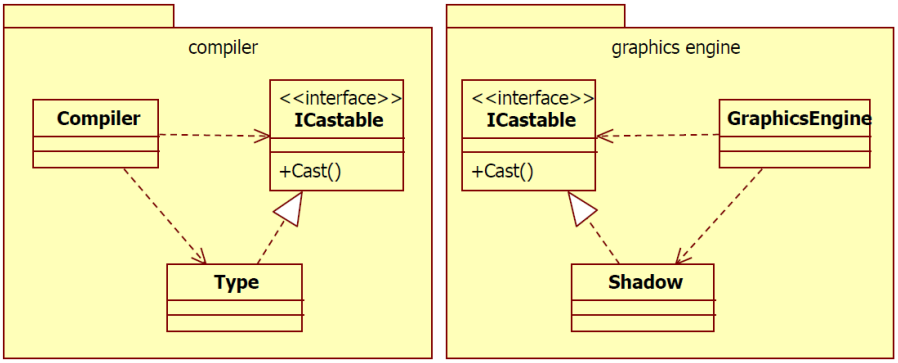
- így a leszármazottak kihasználhatják a közös absztrakció előnyeit
- a kliens osztályoknak pedig nem kell konkrétumokat ismerniük, elég, ha a közös absztrakciótól függenek

"Közös interfészt csak akkor valósítsunk meg, ha a viselkedés is közös!"

- a példával a gond az, hogy a típus és az árnyék közös interfészt implementálnak, de viselkedésükben semmi közös sincs (a típus átkasztolható (cast), a tárgyak árnyékot vethetnek (cast))



- fontos megjegyezni, hogy egy interfészt nem csak függvény szignatúrákat ír elő, hanem azt is meghatározza, hogy melyik függvénynek mi az elvárt viselkedése
- úgy implementálni egy viselkedést, hogy formailag rendben vagyunk, de a viselkedést nem követjük, az a Liskov elv megsértését jelenti
- a példánk megoldása az, hogy a két ICastable interfészt külön-külön definiáljuk, mert formailag ugyan egyformák, de előírt viselkedésükben különböznek

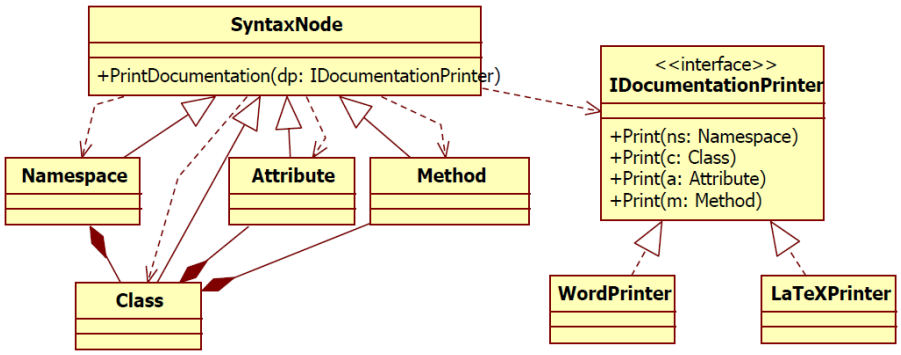


Duck typing

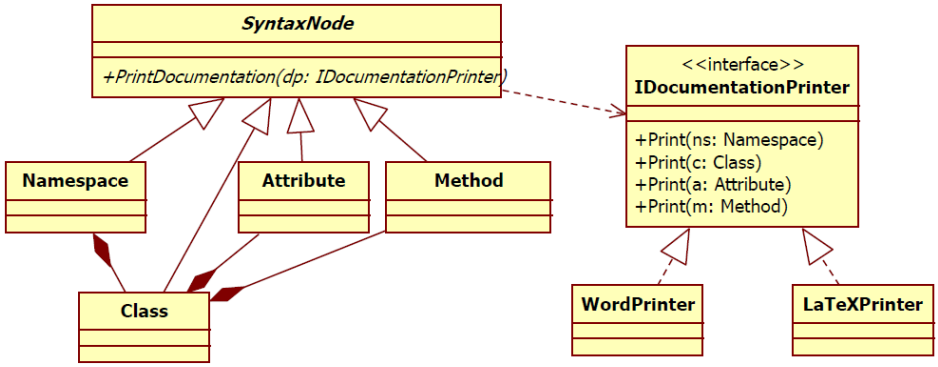
- a duck typing azt jelenti, hogy ha valami úgy mozog, mint egy kacsa, és úgy hápog, mint egy kacsa, akkor az egy kacsa
- tehát ha két típus formailag megegyezik, ugyanolyan függvényeik vannak, akkor azokat tekinthetjük ugyanolyan típusnak
- az erősen típusos nyelvekben, mint a Java, a duck typing nem megvalósítható
- C#-ban is csak a dynamic segítségével valósítható meg a duck typing
- a dinamikus nyelvekben, mint a JavaScript, mindennaposak ezek a megoldások
    - dinamikus nyelvekben előszeretettel használják ezt a megoldást, mert könnyen egységesen kezelhetők a dolgok, de ez nagyon veszélyes vállalkozás, mert a Liskov elv könnyen megsérthető vele
- vigyázzunk arra is, hogy c++ban a templetek a paramétereire előírt kényszerek ugyancsak duck typing-ot követnek

"Egy osztály ne függjön a saját leszármazottjaitól!"

- probléma: az ősosztály függ a leszármazottjaitól, így, ha később új leszármazott jön, azt is bele kell venni az ősbe
    - bővítéskor a meglévő kódhoz is hozzá kell nyúlni, tehát sérül az OCP



- az öröklődés már alapból egy függést jelent, a leszármazott függ az őstől
    - ha az ős is függ a leszármazottjaitól, az egyrészt körkörös függőséghez vezet, másrészt megsérti az OCP levet
- a megoldás az, hogy használjunk polimorfizmust
    - az ősosztályban definiált print függvényt valósítsa meg az összes osztály, és ebben hívják meg a saját print függvényüket (ez a Visitor tervezési minta megvalósítása)



"Protected láthatóságot csak metódusoknál használjunk, az attribútumok mindig privátak legyenek!"

- a protected láthatóság sajátossága
    - a protected láthatóságú függvények nem hívhatóak meg a leszármazottban egy ősosztályon keresztül
    - a protected láthatóságú dolgok csak saját magunkon, vagy valamelyik leszármazottunkon keresztül hívhatóak meg
    - az ősön keresztül, vagy az öröklődési hierarchia egy másik szereplőjén keresztül nem
        - ez azért van így, mert egy osztálynak csak saját magára, illetve leszármazottjaira lehet hatása, az ősre, vagy az öröklődési hierarchia egy másik ágára nem
    - az ősosztály protected láthatóságú függvényét a leszármazottban csak az adott leszármazott típusával megegyező objektumon keresztül lehet meghívni
- attribútumoknál ne használjunk protected láthatóságot, mert a leszármazott elronthatja az őstől megörökölt viselkedést, ha közvetlenül állítgatja az attribútumok értékét
- a szabály tehát az, hogy az attribútumok mindig maradjanak privátok, és amennyiben szükséges, adhatunk hozzájuk hozzáférést protected metódusokon keresztül
- ezeket a metódusokat ne tegyük feleslegesen publikussá, mert az túl sok belső implementációs részletet elárulna, és tele szemetelné az osztály publikus interfészét felesleges függvényekkel

"Az öröklődési hierarchia legyen mély, de legfeljebb hét szintű!"

- minél mélyebb az öröklődési hierarchia, annál finomabban lehet szétválasztani az absztrakciós szinteket
- ennek köszönhetően a közös absztrakciók egységesen kezelhetők
- a hetes szám inkább csak egy iránymutatás: egy ember körülbelül ennyi dolgot tud egyszerre fejben tartani

"Az absztrakt osztályok és interfészek az öröklési hierarchia gyökerében legyenek!"

- levélként nem sok értelmük van, mert absztrakt osztályok és interfészek nem példányosíthatók
- középen még előfordulhatnak, ha vannak további leszármazottjaik
- az itt megfogalmazott szabály a DIP, ISP és SDP egyenes következménye
- kivétel lehet ez a szabály alól az, ha egy olyan könyvtárat készítünk, amelyben kifejezetten támogatni szeretnénk azt, hogy a könyvtár felhasználói kiterjesszék a viselkedést azáltal, hogy implementálják az általunk definiált interfészeket és absztrakt osztályokat
    - ilyenkor ezek az interfészek és absztrakt osztályok ugyan a mi könyvtárunkban levélként jelennek meg, de végső alkalmazásban az öröklési hierarchia gyökerét fogják jelenteni

"Az öröklési hierarchia gyökerében absztrakt osztályok vagy interfészek legyenek!"

- ez a szabály azért fontos mert egy konkrét osztályból később absztrakt osztályt készíteni sokkal nehezebb, mint fordítva
- a szabály különösen fontos akkor, amikor az alkalmazás rétegei közötti interfészt definiáljuk
- a szabály a SAP következménye is, de az ISP, a DIP és az SDP elvek is elősegítik ennek a szabálynak a betartását
- kivételt jelenthet az alól a szabály alól az, ha egy konkrét osztályról tudjuk, hogy nem fog megváltozni
    - ilyen például a szálkezelést biztosító Thread osztály Java-ban, amely önállóan is példányosítható, de akár saját leszármazottat is készíthetünk belőle

"Soha ne vizsgáljuk egy objektum típusát, használjunk helyette polimorfizmust!"

- a dinamikus típusellenőrzés problémája
    - ez nyilvánvalóan sérti az OCP elvet, hiszen, ha bejön egy új típus, akkor át kell írni a típus-ellenőrzést végző függvényt
    - a típusellenőrzés annak a jele is lehet, hogy valamilyen felelősséget rossz helyre osztottunk ki
- használjuk a polimorfizmust, vezessünk be egy metódus a hívott osztályok ősébe, a leszármazottak pedig ezt a metódust írják felül
- kliensként ezt a függvényt hívjuk meg, és a hívott objektum a saját típusa alapján majd úgy viselkedik, ahogy neki szükséges
- a szabály alól azonban lehetnek kivételek
    - például, ha egy konfigurációt mentünk ki, vagy olvasunk be, és ha ehhez reflection-t vagy típustesztelést kell alkalmazni, az teljesen rendben van
    - ugyancsak rendben lehet a típustesztelés akkor, ha az nem sérti az OCP elvet
        - ilyen megoldás például az aciklikus visitor tervezési minta

"Soha ne kódoljuk a típust enum vagy int értékekbe, használjunk helyette polimorfizmust!"

- a dinamikus típusellenőrzés problémája
    - az enum-on keresztüli dinamikus típusellenőrzés is sérti az OCP elvet, hiszen, ha az enum egy újabb taggal bővül, akkor át kell írni a meglévő kódokat
- használjuk a polimorfizmust, vezessünk be egy metódus a hívott osztályok ősébe, a leszármazottak pedig ezt a metódust írják felül
- kliensként ezt a függvényt hívjuk meg, és a hívott objektum a saját típusa alapján majd úgy viselkedik, ahogy neki szükséges

"Ne készítsünk függvényeket a típusok, illetve a képességek megkülönböztetésére, használjunk helyettük polimorfizmust!"

- a dinamikus típusellenőrzés problémája
    - a típuslekérdező függvények írása is sérti az OCP elvet
    - hiszen, ha bejön egy új típus, akkor az összes helyen, ahol a típusra rákérdeztünk, át kell írni a kódot
- a típuslekérdezések a TDA elvet is sértik, ami azt jelzi, hogy a felelősségeket rosszul osztottuk ki

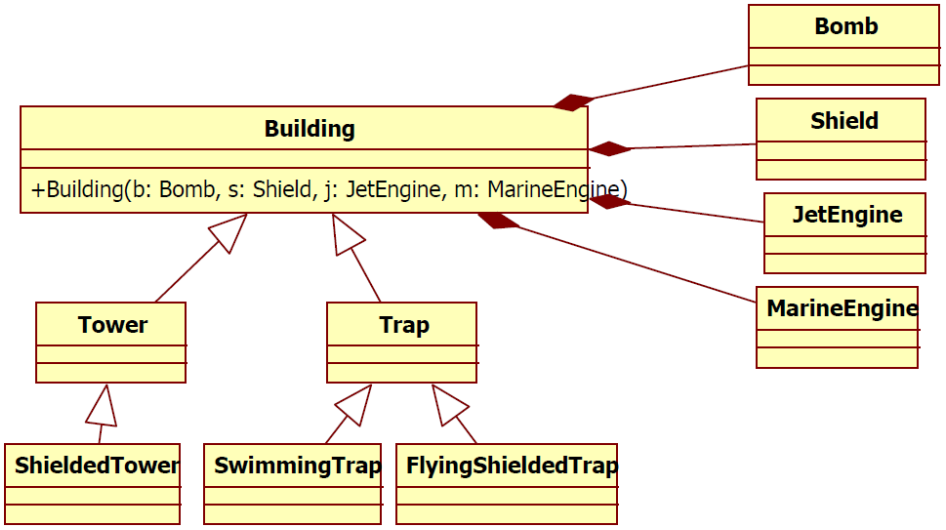
- a kliensnek ugyanis nem szabad ellenőriznie a szerver típusát, illetve belső állapotát, hanem egyszerűen csak közölnie kell a szerverrel, hogy milyen műveletet szeretne rajta végrehajtani
- a szerver működése pedig polimorfizmussal kijön, a saját típusa alapján, és a megfelelő működést is a szerver hajtja végre a saját belső állapota alapján

"Ne keverjük össze a leszármazottakat az objektumokkal! Vigyázzunk azokkal a leszármazottakkal, amelyekből csak egyetlen példányt hozunk létre!"

- győződjünk meg arról, hogy a leszármazottak ténylegesen adnak-e hozzá viselkedést az őshöz, vagy módosítják-e az ősben definiált viselkedést
- amennyiben nem, akkor nincsen szükség leszármazott osztályokra
- ilyenkor elegendő az ősosztályból több példányt létrehozni különböző belső állapotokkal
- vizsgáljuk meg azt is, hogy a leszármazottak viselkedése mennyire hasonlít egymásra
- ha ez a viselkedés általánosítható, akár újabb attribútumok bevezetésével az ősben, akkor megint csak felesleges megtartani a leszármazottakat
- ha tehát a leszármazottaknak semmilyen plusz viselkedésük nincs az őshöz képest, akkor eliminálju̇k őket, és amennyiben tényleg szükség van rá, egy plusz állapotot felvehetünk az ősbe (sokszor azonban még erre sincs szükség)

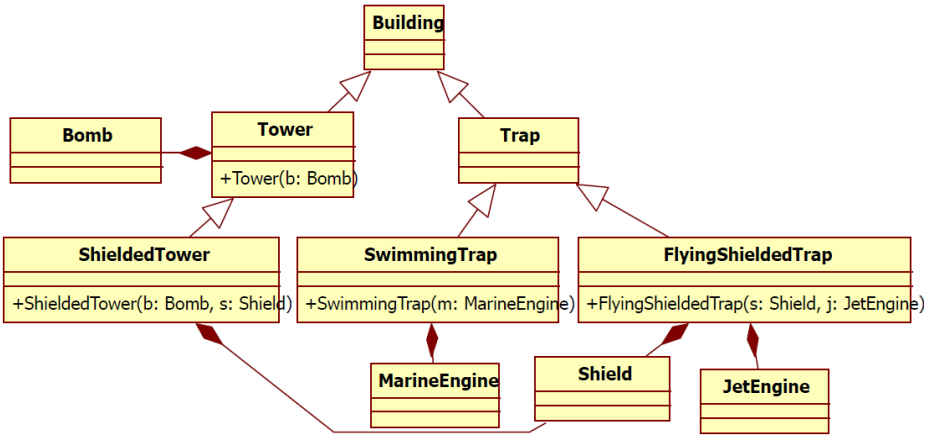
Dinamikus komponensek problémája

- a célunk az, hogy kisebb komponensekből nagyobb komponenseket építsünk
- jelen esetben egy Tower Defence jellegű játékban különböző képességeket szeretnénk létrehozni
- a nehézség az, hogy a nagyobb komponenst alkotó kisebb komponensek dinamikusan cserélgethetők
- például, ha egy toronyhoz bombát rendelünk akkor tud lőni, ha pajzsot rendelünk, akkor tovább bírja a támadást, ha valamilyen hajtóművet rendelünk hozzá, akkor tud úszni vagy repülni stb...
- és ezek a képességek tetszőleges módon kombinálhatók és cserélgethetők



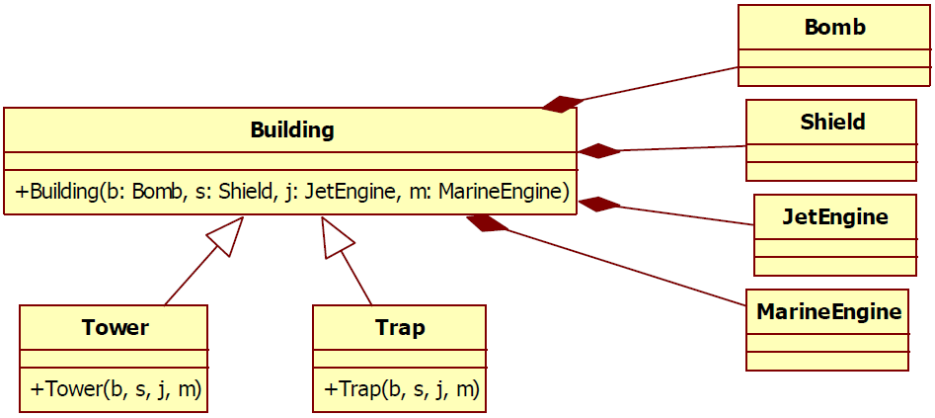
"A statikus szemantikát és kényszereket a modell struktúrájába építsük be! (ha kevés a helyes kombináció)"

- a statikus elemek itt azok, amelyek dinamikusan nem cserélgethetők
- ha a komponensek csak nagyon limitált módon kombinálhatók össze, és egy-egy kombináció dinamikusan nem cserélgethető, akkor célszerű minden lehetséges kombinációra egy önálló osztályt definiálni
- ha sikerül ennyire megkötni a fejlesztők kezét, akkor egyáltalán nem tudnak hibás kombinációkat létrehozni, a felépített modell mindig korrekt lesz



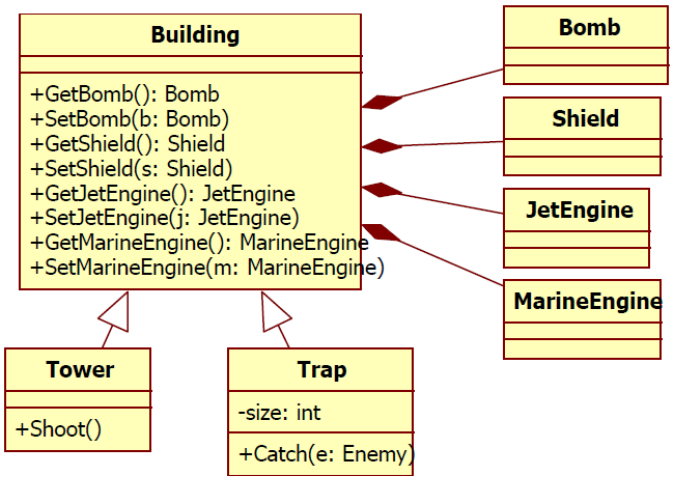
"A statikus szemantikát és kényszereket a konstruktorba építsük be! (ha túl sok a helyes kombináció)"

- ez azt jelenti, hogy a konstruktor akár hibás kombinációkkal is meghívható, de a kényszerek ellenőrzése miatt a konstruktor kivételt fog dobni, a hibás objektum pedig nem jön létre
- ennek a megoldásnak a segítségével kevés osztállyal sok kombinációt le lehet fedni, azonban a hibás kombinációk már fordítási időben nem ismerhetőek fel, csak futási időben fog kiderülni, ha valami baj van



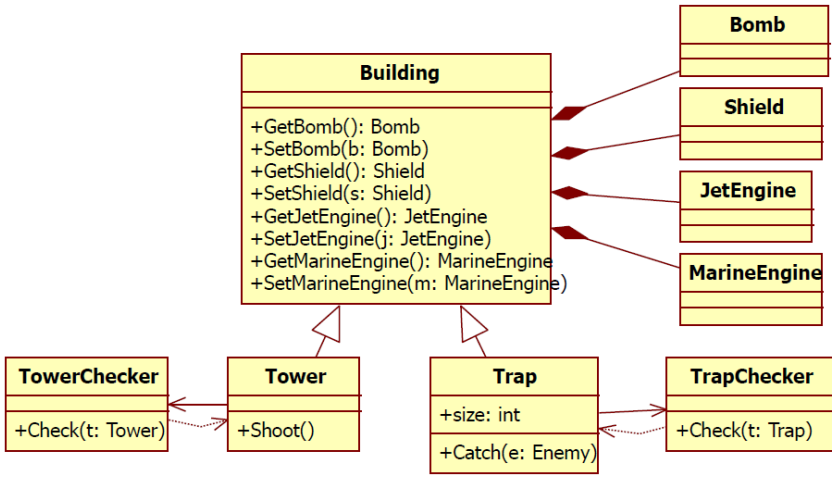
"A dinamikus szemantikát és kényszereket viselkedésként implementáljuk!"

- ha vannak dinamikusan cserélgethető komponensek, azokat már nem lehet a konstruktorban ellenőrizni, hanem a kényszereket a metódusokban kell betartatni
- a komponensek ilyenkor dinamikusan cserélgethetőek, és a célzott viselkedés esetén ellenőrizzük, hogy az végrehajthatóak-e (például a torony példában lövés esetén ellenőrizzük, hogy van-e bomba)



"A gyakran változó dinamikus szemantikát és kényszereket külső viselkedésként implementáljuk!"

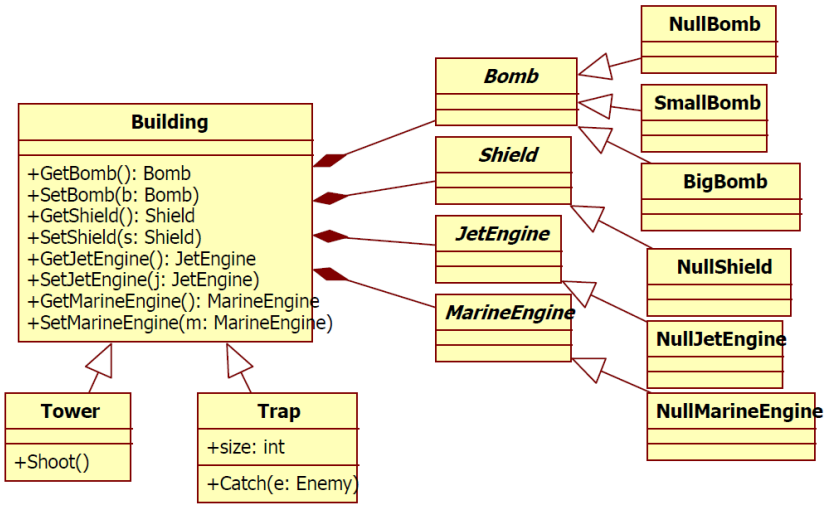
- ha tehát a működés feltételei sem határozhatóak meg statikusan, azok is dinamikusan változhatnak, akkor a kényszerek ellenőrzését ki kell szervezni az osztályból
    - ebben segíthetnek az olyan tervezési minták, mint a Strategy, a Command, a Chain of Responsibility és a Visitor
- a torony példájában ez azt jelenti, hogy egy lövés esetén a torony helyes állapotát egy külső osztály ellenőrzi
    - erre akkor lehet szükség, ha a lövés körülményei dinamikusan változhatnak
    - mondjuk például, ha leszáll a köd, akkor csak bizonyos tulajdonságú lövedékekkel lehet lőni
    - a köd viszont bármikor felszállhat, vagy mozoghat a pályán össze-vissza, így ennek a kényszernek az ellenőrzése nem a torony feladata



"Az opcionális elemeket tartalmazásként implementáljuk, ne öröklődéssel!"

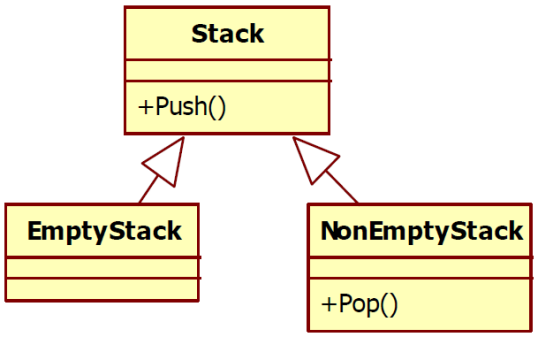
- ez a szabály következik abból a szabályból, hogy próbáljuk meg elkerülni az osztályok számának kombinatorikus robbanását
- amennyiben egy opcionális elem hiányzik, célszerű ott nem null értéket tárolni, hanem érdemes megfontolni a Null object tervezési minta használatát
    - az egyszerű nullértékekkel az a baj, hogy folyamatosan ellenőrizgetni kell őket feltételekben

    - a null object ezzel szemben egy strázsa jellegű megoldás, ugyanis ugyanolyan interfészű, mint egy normál objektum, azonban működésben nem csinál semmit, vagy valamilyen alapértelmezett viselkedést produkál
    - így, ha egy opcionális objektum helyett egy null object-re mutatunk, akkor mindenféle feltételellenőrzés nélkül meghívhatjuk rajta a függvényeket, és megkapjuk az alapértelmezett viselkedést
    - valódi működés csak akkor történik, ha a tényleges objektumon hívjuk a függvényeket
- a példánkban ez azt jelenti, hogy például amíg a toronyhoz nem rendelünk tényleges lövedéket, addig egy NullBomb tölti be a lövedékek szerepét
    - a NullBomb ugyanolyan függvényekkel rendelkezik, mint egy sima Bomb, de valójában nem csinál semmit



"Ne keverjük össze a statikus és dinamikus kényszereket!"

- ne definiáljunk külön osztályt a veremnek és az üres veremnek, mert a verem kiürülésekor, vagy az üres verem feltöltésekor az objektumnak típust kellene váltania
    - de az objektumok típusváltása nem lehetséges
    - a probléma forrása az, hogy a verem üressége nem egy statikus tulajdonság, hanem a dinamikus viselkedés eredménye



- a statikus kényszereket a modell struktúrájában vagy a konstruktorokban implementájuk
- a dinamikus kényszereket pedig a viselkedésben, tehát a metódusokban

"Ha reflection-ra van szükségünk modellezzünk osztályokat, ne objektumokat!"

- ha reflection-t használunk, az általában annak a jele, hogy nem a megfelelő absztrakciós szinten modellezünk
- tehát objektumokat próbálunk meg modellezni ahelyett, hogy osztályokat terveznénk
- ha tehát reflection-ra építünk, vizsgájuk meg, hogy tényleg szükség van-e rá, vagy esetleg eggyel magasabb absztrakciós szintre kellene lépni a modellezéssel
- természetesen a reflection-nak is megvan a maga helye
    - ha például ki kell menteni az objektumokat fájlokba, vagy adatbázisba, vagy konfigurációs beállításokat kell menteni, vagy visszatölteni, akkor célszerű reflection-t használni

"Ha az ős működését üres implementációval írjuk felül, akkor hibás az öröklési hierarchia!"

- ha az ős függvényét egy üres implementációval írjuk felül, akkor nem tartjuk be az őstől elvárt viselkedést, vagyis megsértjük a Liskov-féle helyettesítési elvet
- ilyen helyzet fordulhat elő akkor is, ha az ősosztály nagyon sokfajta viselkedéskombinációt implementál, a leszármazottak pedig üres implementációkkal kilövöldözik a számukra irreleváns viselkedéseket
    - ez a megoldás is sérti a Liskov-elvet
- ilyenkor tipikusan arról van szó, hogy az öröklési hierarchiában túl magasra kerültek olyan viselkedések, amelyek egyes leszármazottak számára nem kívánatosak
- ez azt jelenti, hogy összekeveredett az öröklési hierarchia, érdemes elgondolkodni azon, hogy felcseréljük benne, az osztályok sorrendjét

"Törekedjünk újrahasznosítható API írására, ne csak újrahasznosítható osztályokéra!"

- az API írási szabályok sokkal szigorúbbak, mint a tervezési heurisztikák, és céljuk, hogy biztosítsák az API hosszútávú stabilitását
- ha ugyanis publikálunk egy könyvtárat, annak van egy API-ja, amit ha mások is elkezdenek használni, azt később, már nagyon nehéz lesz megváltoztatni
    - emiatt egy API mindig nagyon stabil
    - alaposan meg kell gondolni, hogy mit publikálunk, mert azt később már visszavonni nem tudjuk
- nyilvánvalóan a mindennapi munkában nem kell ezeket a szabályokat tartani, de nem árt gondolni rájuk, hátha az általunk írt modulok később még más projektekben is felhasználhatóak lehetnek
- törekedjünk tehát a minél általánosabb megoldásra, de azért tartsuk be a rendelkezésre álló pénz- és időkereteket

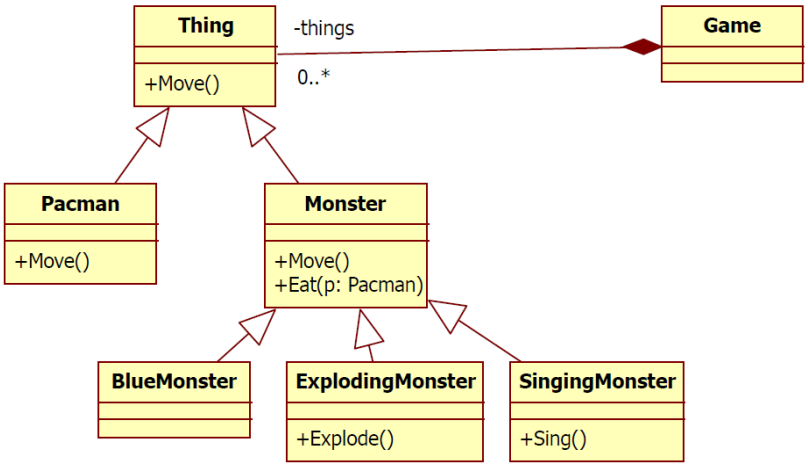
"Ha többszörös öröklődésre van szükségünk, gondoljuk át még egyszer a terveket!"

- a többszörös öröklés jelezheti, hogy valahol öröklést használtunk tartalmazás helyett, vagy rosszul vettük fel az öröklési hierarchiákat, de a legfőbb probléma a több ágon is megörökölt adat és viselkedés kezelése
- ha többszörös öröklést kell használnunk, érdemes elvégezni azt a gondolatkísérletet, hogy valamit elrontottunk, majd próbáljuk bebizonyítani ennek ellenkezőjét

    - ha sikerül, akkor valóban szükség van a többszörös öröklésre, ha nem, akkor próbáljuk meg eliminálni
- a többszörös öröklés azért is problémás, mert nagyon kevés programnyelv támogatja
    - az olyan nyelvekben, ahol nincs meg ez a támogatás, tipikusan csak kódduplikációval lehet megvalósítani, ez pedig sérti a DRY elvet

Heterogén kollekció problémája - leszármazott egyedi függvényének meghívása heterogén kollekcióból

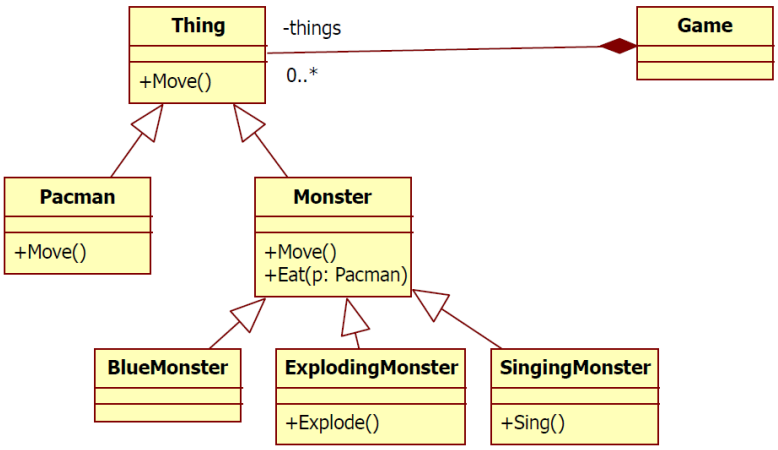
- probléma: a Game osztály egy gyűjteményben tárolja a játék szereplőit Thing típusként
    - az egyes szereplőknek van saját függvényük (például az egyik tud énekelni a másik pedig fel tud robbanni)
    - a kérdés az, hogy hogyan érjük el ezeket a függvényeket a Game osztályból



- az egyik megoldás az, hogy ezeknek a viselkedéseknek találunk egy közös őst, és azt felvisszük a dolog ősbe
    - a probléma csupán az, hogy hogyan nevezzük el, hogy mit tegyünk akkor, ha a két függvény szignatúrája különbözik, illetve biztosan szüksége van-e az összes leszármazottnak erre a metódusra (nehogy valakit olyan viselkedésre kényszerítsünk, amihez neki semmi köze)
    - ez tehát nem tűnik járható útnak
- egy másik lehetséges megoldás az, hogy az osztályok megmondják, hogy milyen képességeik vannak (például tudnak-e robbanni, vagy tudnak-e énekelni)
    - korábban azonban láttuk, hogy ez olyan, mint a típusellenőrzés, sérti az OCP elvet

Heterogén kollekció problémája - adott függvény hívása csak egyes típusokon a heterogén kollekcióban

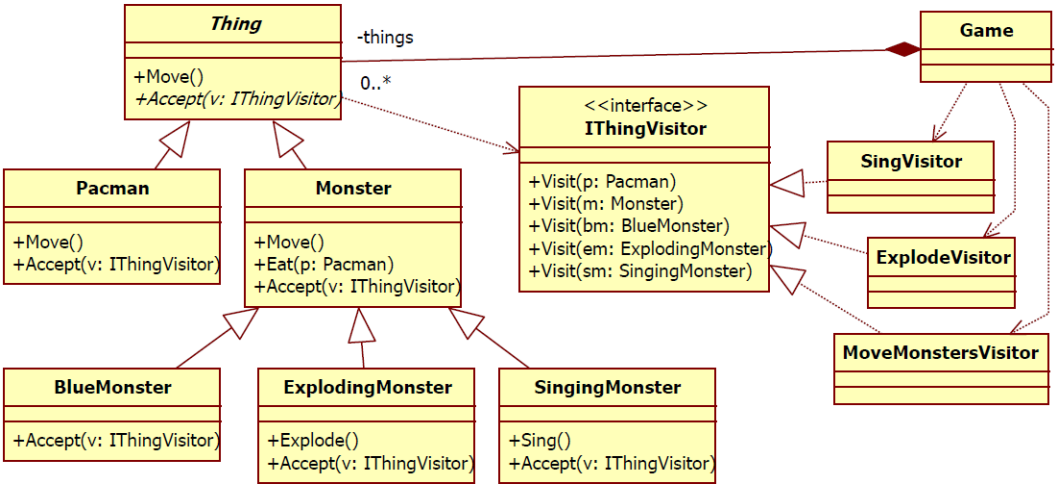
- probléma: a Game osztály egy gyűjteményben tárolja a játék szereplőit Thing típusként
    - minden leszármazottnak van Move függvénye
    - a kérdés az, hogy hogyan érjük el a Game osztályból, hogy csak a Pacman lépjen, a szörnyek ne?



- felmerülhet ötletként, hogy a szörnyeknek legyen egy külön gyűjteményünk, ez azonban sérti az OCP elvet
- egy másik lehetséges megoldás a típusok ellenőrzése, azonban ez is sérti az OCP elvet

Heterogén kollekció megoldás

- a heterogén kollekció problémáira a Visitor tervezési minta lehet megoldás
- a Visitor interfész definiál egy függvényt minden egyes meglátogatható elemre
- és minden egyes elem a saját Accept függvényét úgy implementálja, hogy a saját magának megfelelő Visit függvényt hívja vissza
    - így implementálhatóak azok a Visitorok, amelyek csak a számukra érdekes elemekre reagálnak
- ha egy ilyen Visitort végig küldünk az összes elem Accept függvényén, akkor a Visitor csak a számára releváns elemekre fog hatni
    - ez a példa jól mutatja a Visitorokba kiszervezett viselkedés erejét
- sajnos azonban ez a megoldás sem teljesen tökéletes, hiszen az OCP elv itt is sérül
    - ha ugyanis újabb dolgot veszünk fel a játékba, a Visitor interfészt ki kell bővíteni a neki megfelelő függvénnyel, és minden olyan Visitort is át kell írni, amely ezt az interfészt implementálja
    - ezen a problémán azonban könnyen segíthetünk az aciklikus Visitor tervezési minta használatával, ugyanis az a változat nem sérti az OCP elvet



Heurisztikák szigorúsága

- fontos hangsúlyozni, hogy a heurisztikákat nem lehet mindig 100%-osan betartani
- sőt, egyes heurisztikák ellent is mondhatnak egymásnak
- érdemes inkább úgy gondolni rájuk, mint a kalóz-kódra, ezek inkább csak irányelvek, mint valódi szabályok
- ha mérnökök vagyunk, a célunk, hogy egy működő alkalmazást rakjunk össze
- ebbe nem fér bele az, hogy órákon vagy napokon keresztül objektumorientált tervezési vallásháborúkat vívjunk egymással
- próbáljuk meg betartani az itt tanult szabályokat, de ha nem találunk jobb megoldást, nyugodtan sértsük meg őket
- ez a megjegyzés azonban nem jogosít fel arra, hogy teljesen figyelmen kívül hagyjuk őket
- a lényeg, hogy a végén a szoftver működjön, és lehetőleg könnyű legyen tovább fejleszteni

Refaktorálás

Refaktorálás fogalma

Refaktorálás lényege

- már van egy működő kódunk
- milyen módon javíthatunk annak minőségén

Mi a refaktorálás?

- Refaktorálás:
  - a külsőleg megfigyelhető viselkedés megtartásával a szoftver belső struktúrájában végrehajtott olyan változtatás, amely által a szoftver kódja könnyebben érthetővé és olcsóbban módosíthatóvá válik (Martin Fowler)
- Refaktorálni:
  - a szoftver átstrukturálása refaktorálások sorozatával úgy, hogy a külsőleg megfigyelhető viselkedés nem változik

Mikor refaktorálunk?

- új funkciót szeretnénk hozzáadni a szoftverhez, de azt nehéz megtenni, mert arra a szoftver nem lett felkészítve (maga az új funkció hozzáadása nem jelent refaktorálást)
- bug-ot kell kijavítanunk, de a kód nehezen érthető, és nehéz megtalálni a hibát
- amikor kódszemlét tartunk, és valami nehezen olvasható és nehezen érthető kódot látunk

Refaktorálás szabályai

- mivel a refaktorálás során a külső viselkedés nem változhat ezért szükségünk van egy alapos tesztkészletre, amivel a külső viselkedés változatlanságát tudjuk tesztelni
- a refaktorálást mindig kis lépésekben érdemes elvégezni, és közben folyamatosan ellenőrizni kell, hogy a külső viselkedés nem változik
- merjünk változtatni, ugyanis a tesztek segítenek abban, hogy a viselkedés ne romoljon el

Refaktorálás lépései

1. Legyen alapos tesztkészlet
2. A régi kód menjen át a teszteken
3. Apró változtatás végrehajtása
4. Az új kód is menjen át a teszteken
5. Ismétlés az 1. lépéstől

Refaktorálás előnyei

- javítja a szoftver terveit
  - a szoftver belső struktúrája ugyanis folyamatosan romlik
  - minden egyes újabb változtatás, minden egyes újabb követelmény implementálása csak ront a helyzeten

        - a refaktorálás segít abban, hogy a struktúra helyre álljon
- a kód olvashatóbbá és érthetőbbé válik
    - ez azért is fontos, mert a fejlesztők többet olvassák a kódot, mint írják
    - ha a kód könnyen érthető, az nagyon sok időt megtakarít a fejlesztő számára
    - itt érdemes nem csak más fejlesztőkre gondolni, hanem saját magunkra is, amikor évek távlatából visszaolvassuk a kódunkat
- segít a bugok megtalálásában
    - egyrészt meg kell érteni a kódot ahhoz, hogy refaktoráljuk, és már a megértés során felfedezhetünk hibákat
    - másrészt a refaktorálás elkezdéséhez muszáj teszteket írnunk, és ezek a tesztek is felderíthetnek korábban megbújó hibákat
    - sajnos azonban a refaktorálás során is vihetünk be újabb hibákat, de azért van szükség a sok-sok tesztre, hogy ezek minél hamarabb kiderüljenek
- segít, hogy felgyorsítsuk a fejlesztést
    - rövidtávon lehet, hogy több időt eltöltünk a refaktorálással, de hosszabb távon mindenképpen vissza fogjuk nyerni az eltöltött időt, hiszen a későbbi új funkciókat már egy jól strukturált kódban lehet elhelyezni

Nehezen refaktorálható programok

- adatbázis + komponensek közti interfészek
    - ezek általában stabilak, és sok minden függ tőlük
    - így egy-egy változtatásnak sok mindenkire hatása van
    - ezért is fontos az, hogy már a legelején jól tervezzük meg őket
    - ha később mégis refaktorálni kell, akkor adatbázisoknál valamilyen adatmigrálásra lesz szükség, az interfészeknél pedig egy darabig támogatni kell a korábbi verziót is

Büdös kód (code smell)

Büdös kód

- *"A büdös kód egy olyan felületi tünet, amely valamely mélyebb tervezési problémát jelez"* (Martin Fowler)
    - ennek segítségével lehet felismerni, ha valamit refatorálni kell
    - ha a büdös kódot sikerül felismerni az már félsiker
    - a javítás ugyanis triviális szokott lenni
    - fontos hangsúlyozni, hogy a büdös kód nem bug
    - a szoftver működhet teljesen jól, akár hibátlanul, a büdös kód ugyanis csak annyit jelez, hogy nehéz olvasni és nehéz továbbfejleszteni a szoftvert
- mindenesetre a büdös kód arra utal, hogy a szoftver tervei gyengélkednek, lelassulhat a fejlesztés, és megnő a későbbi bugok bevitelének kockázata
- a következőkben ilyen code smell-eket és az ezek esetén használható refaktorálási lehetőségeket fogjuk áttekinteni

Duplikált kód

- *"Ugyanaz vagy nagyon hasonló kód ismétlődik több helyen"*
- ez általában annak a jele, hogy megsértjük a TDA elvet, de DRY elvet mindenképpen

- a duplikált kódnál a fő gondot az okozza, hogy módosítás esetén könnyű hibát véteni, mert nem biztos, hogy az összes előfordulást megtaláljuk
- a duplikált kód refaktorálása történhet azáltal, hogy a kódot kiemeljük egy önálló függvénybe, vagy bonyolultabb kódrészlet esetén akár egy önálló osztályba is

Hosszú metódus

- *"A függvény kódja túl hosszú: túl sok feltételes ág illetve ciklus található benne"*
- ilyenkor a metódus kódját nehéz megérteni, nehéz módosítani
- a refaktorás az lehet, hogy kommentek mentén feldaraboljuk a függvényt
    - a magyarázó kommentek ugyanis egy-egy bonyolultabb rész határát jelzik
- egy másik megoldás lehet a függvény feldarabolása a feltételes elágazások és ciklusok blokkjai mentén
- megoldás lehet még egy metódus objektum bevezetése, amely azt jelenti, hogy a függvényben implementált algoritmust egy önálló osztályba szervezzük, ahol a hosszú függvény blokkjaiból önálló metódusok lesznek

Hosszú paraméter lista

- *"A függvénynek túl sok (háromnál több) paramétere van"*
- ilyenkor nehezebb megérteni, hogy melyik átadott paraméter mit jelent
- tesztelni is nehezebb a függvényt, mert túl sok bemeneti érték-kombináció lehetséges
- refaktorálásként feldarabolhatjuk a függvényt több kisebb függvényre, vagy a hosszú paraméterlistából egy önálló objektumot lehet létrehozni, és a függvények között nem a sok-sok paramétert, hanem ezt az egy db függvényt kell cserélgetni
- ez azért is előnyös lehet, mert a paramétereket leíró osztály később akár saját felelősségeket is kaphat, és egy önmagában erősebb osztállyá léphet elő

Nagy osztály

- *"Az osztálynak túl sok metódusa van"*
- ez jelentheti azt, hogy az osztálynak túl sok felelőssége van, sérül a SRP
- annak a jele is lehet, hogy isten-osztállyal van dolgunk
- az is probléma, hogy a kliensek valószínűleg nem használják fel a nagy osztály összes függvényét
- így a kliensek olyan metódusoktól is függenek, amelyeket nem használnak, vagyis sérül az ISP
- refaktorálásként feldarabolhatjuk az osztályt több kisebb osztályra, de ha ez nem működik használhatjuk az ISP-t
- egy másik lehetséges megoldás, hogy öröklési hierarchiát készítünk az osztályból
- megoldás lehet az is, hogy az osztály felelősségeit a metódusok átrendezésével szétosztjuk az osztályok között

Divergent change

- *"Felhasznált technológiánként más és más osztályt kell használnunk"*
- például különböző grafikus technológiáknál, vagy különböző adatbázis drivereknél más és más módon kell elvégezni a műveleteket
- ilyenkor egy-egy változtatás a kód sok más területére is hatással lehet

- a refaktorálási megoldás az lehet, hogy a technológiák közös részét egy közös stabil absztrakt csomagba kiszervezzük, ezt használja a kód többi része, a konkrét technológiai implementációk pedig ezeket az absztrakciókat implementálják
- a Bridge tervezési minta is ezt alkalmazza

Shotgun surgery

- *"Egy változtatás sok más osztályban apró változtatásokat indukál"*
- ez tipikusan akkor fordul elő, ha konstansokat, sztring literálokat beleégetjük a kódba, értékük megváltoztatásához pedig az egész kódot át kell fésülni
- ilyenkor sérül a DRY elv is
- egy refaktorálási megoldás lehet az, hogy ezeket a konstansokat és literálolak elnevezzük, és egy helyen összegyűjtjük
- sőt, ezeket érdemes erőforrás fájlokba kiszervezni, hogy később a többnyelvűség támogatása könnyen megoldható legyen

Feature envy

- *"Egy osztály túlságosan érdeklődik egy másik iránt, vagyis túl sok függvényt hív a másikból"*
- ez annak a jele lehet, hogy a felelősségeket rosszul osztottuk ki, és az egyes metódusok nem a megfelelő osztályhoz tartoznak
- emiatt túlságosan magas a csatolás a két osztály között
- a megoldás az lehet, hogy azokat a függvényeket, melyeket a szerverből sokat hívunk, átrakjuk a kliensbe
    - így a kliensen belül fog nőni a kohézió, az osztályok között pedig csökken a csatolás
- az is elképzelhető, hogy a kliensnek csak néhány függvénye hívja a szerverosztály sok-sok függvényét
    - ilyenkor az is megoldás lehet, hogy a kliens függvényei rakjuk át a szerverbe
    - ezáltal a szerverben nő a kohézió, és csökken a két osztály közti csatolás
- a lényeg az, hogy az egymást használó, és együtt változó dolgok egy helyre kerüljenek

Data clups (adatcsomósodás)

- *"Ugyanaz a paramétercsoport ismétlődik több metódushíváson keresztül"*
- az adatelemek természetesen csoportosulnak
    - például egy személy neve, életkora, lakcíme lehet egy ilyen csoportosulás
    - sok ilyen paramétert nagyon körülményes átadogatni függvények között
- egy ilyen megoldás tipikusan azt jelenti, hogy nem objektum orientáltan terveztünk a felelősségek figyelembevételével, hanem procedurálisan, adatközpontú tervezést végeztünk
- a megoldás az lehet, hogy ezeket a paramétereket építsük be az osztályokba mezőként
    - így minden metódus eléri őket, és nem kell folyton paraméterként átadogatni
- egy másik lehetséges refaktorálás az lehet, hogy egy paraméterobjektumot hozunk létre, vagyis készítünk egy olyan osztályt, amelynek attribútumai lesznek a csoportosuló paraméterek
    - a metódusok között ezután elegendő ennek az osztálynak egy példányát átadogatni

- érdemes azt is megnézni, hogy kik hívogatják ezeket a függvényeket, mert előfordulhat, hogy közülük pár felelősség átrakható a paraméterobjektumot leíró osztályba, így az nem csak egy egyszerű adatosztály lesz, hanem felelősségekkel is fel tudjuk ruházni

Primitive obsession (primitív típusokhoz való ragaszkodás)

- *"Adatok primitív típusokban vannak tárolva osztályok helyett"*
- ez annak a jele lehet, hogy a tervezéskor nem az objektum orientált tervezési elveket követtük, a megoldásunk így nehezen bővíthető, a primitív adatokhoz pedig nem lehet felelősségeket rendelni
- refaktorálásként lecserélhetjük a primitív adatcsoportosulásokat osztályokra, vagy ha éppen tömbben vannak tárolva az adatok, akkor a tömb helyett készítsünk osztályt
- ha a primitív elemek mellett típusra utaló elemek is vannak, akkor a refaktorálás során létrejövő osztályok között öröklődést is definiálhatunk

Switch statements

- *"A kódban túlságosan sok a feltételes elágazás"*
- ilyen például a hosszú if-else elágazás, a null értékek folyamatos ellenőrizgetése, vagy akár egy sok ággal rendelkező switch utasítás
- az ilyen megoldás sokszor kódduplikációhoz vezet, és tipikusan azt jelzi, hogy a felelősségek rosszul vannak kiosztva
- refaktorálásként érdemes megfontolni a polimorf viselkedést a feltételes elágazások helyett
- vagyis ahelyett, hogy a feltételeket ellenőriznénk, hívjunk meg egy virtuális függvényt, a leszármazottak pedig ezt definiálják felül a megfelelő viselkedéssel
- a null ellenőrzések helyett pedig használhatjuk a null object tervezési elvet

Parallel inheritance hierarchies (párhuzamos öröklődési hierarchiák)

- *"Ha egy osztályból új leszármazott készül, akkor egy másikból is kell leszármazottat készíteni"*
- ilyenkor tipikusan a két öröklődési hierarchiában az osztályneveknek azonos elő- vagy utótagjai vannak
- mivel ennél a code smell-nél egy olyan változtatás, amely új osztályt hoz létre a rendszer más részeiben is változtatást indukál, vagyis ott osztályokat kell létrehozni, ezért ez a code smell a shotgun surgery-nek egy speciális esete
- a megoldás az lehet, hogy próbáljuk meg összevonni a két hierarchiát egybe
- a szabályt azonban nem kell betartanunk, ha a párhuzamos öröklődési hierarchiák tudatos tervezői döntések eredményei, például a Bridge tervezési minta

Lazy class (lusta osztály)

- *"Az osztály túl kevés dolgot csinál"*
- ilyen osztályt általában nem szándékosan hozunk létre a tervezés során, hanem refaktorálási lépések eredményeként keletkezhet (például, ha a felelősségeit elvesszük, és más osztályhoz rendeljük hozzá)
- egy ilyen osztályt felesleges karban tartani, legjobb, ha eliminájuk
- ha esetleg maradt benne még néhány funkció, azokat beépíthetjük a hívó függvényekbe

- ha esetleg az öröklési hierarchiában keletkezik ilyen osztály, akkor azt össze lehet vonni az ősével, vagy a leszármazottjaival

Speculative generality (spekulatív általánosság)

- *"Az osztály feleslegesen túl nehézsúlyú"*
- ez tipikusa annak az eredménye, hogy túlterveztük a rendszert
- olyan követelményváltozásra akartunk vele felkészülni, amely nagy valószínűséggel nem fog bekövetkezni
- olyan tervezési mintákat alkalmaztunk, melyek karbantartása feleslegesen sok overhead-et ró a fejlesztésre
- egy ilyen teher cipelése nagyon megdrágítja a fejlesztést
- egy ilyen helyzet a YAGNI elv megsértését eredményezi
- refaktorálásként eliminájuk a nehézsúlyú dolgokat, egyesítsük az öröklési hierarchiákat, és a nem használt elemeket, például paramétereket töröljük

Temporary Field (ideiglenes mező)

- *"Egy attribútum csak bizonyos esetekben van használva"*
- ilyenek az objektum-szintű ("globális") változók, melyek nem részei az objektum állapotának, hanem az objektumon belül, a függvényhívások között a paraméterátadások számát hivatottak csökkenteni
- tipikusan ilyenek azok a metódusok, amelyek valamilyen bonyolult algoritmus implementálnak, és közöttük információ-cserére van szükség
- a probléma ezzel a megoldással az, hogy nem minden függvény használja ezeket az attribútumokat, így az osztályon belül alacsony a kohézió
- egy másik problémát is okozhatnak ezek a mezők: mivel ezek egy algoritmus belső állapotát tárolják, az algoritmusból egyszerre csak egy futtatható az objektumon (mondjuk egy szélességi keresés egy gráfban)
    - ez azt jelenti, hogy az objektumot nem lehet egyszerre több szálról használni, és az objektum nem is tehető szálbiztossá
- refaktorálásként azt tehetjük, hogy ezeket az attribútumokat, és a hozzájuk tartozó algoritmust egy külön osztályba szervezzük ki
- valahányszor pedig az algoritmust futtatni szeretnénk, ebből az osztályból kell egy új objektumot létrehoznunk
- mivel minden egyes hívás során új példány jön létre, az algoritmus futása szálbiztos lesz
- az algoritmushoz tartozó osztály kohéziója magas lesz, hiszen az eddig ideiglenesen használt változókat az algoritmus sokat fogja használni
- az eredeti osztály kohéziója is nő, mert kivettük belőle az ideiglenes mezőket

Message chains

- *"Hosszú metódushívási láncok a kódban"*
- ezek a Demeter törvény megsértését jelentik
    - ilyenkor a kliens a lánc mindegyik elemétől függ, bármelyik megváltozása hatással lehet rá
- egy lehetséges refaktorálás az lehet, hogy a függvényeket mozgassuk át az osztályok között, hátha a felelősségek rosszul vannak kiosztva

- egy másik lehetséges refaktorálási lépés az lehet, hogy minden egyes láncszemnél a lánc maradékát egy deleáló függvényben elrejtjük
    - ez a megoldás azért előnyös, mert így mindenki csak a közvetlen szomszédját ismeri
    - a megoldás hátránya az lehet, hogy a delegáló függvények száma elburjánozhat
    - ha túl sok ilyen delegáló függvény keletkezne, akkor inkább sértsük meg a Demeter törvényt, de inkább gondolkodjunk el azon, hogy a felelősségek biztosa jól vannak-e kiosztva

Middle man

- *"Egy osztály túl sokat delegál egy másik osztály felé"*
- ezt a köztes delegáló osztályt felesleges fenntartani
- refaktorálásként töröljük ezt a köztes osztályt, és hívjuk a cél osztályt közvetlenül
- ha a köztes osztály néhány plusz köztes műveletet is végez, akkor inline beépíthetjük ezeket a műveleteket a hívóba, hogyha ez kódduplikációt eredményez, akkor a hívott osztályba
- természetesen arra ilyenkor figyeljünk, hogy a célosztályt más nem használja, és ezáltal nem rontjuk-e el az ő működését
- egyes esetekben megoldás lehet az is, ha a delegálást öröklődéssel váltjuk ki
- ezzel azonban óvatosan bánjunk, győződjünk meg arról, hogy valóban helyesen használjuk-e az öröklődést, hiszen a heurisztikánk azt mondja, hogy öröklődés helyett általában delegálást érdemes használni

Inappropiate intimacy

- *"Egy másik osztály privát tagjainak közvetlen elérése"*
- ilyenkor sérül az egységbezárás elve, és túlságosan magas lesz a csatolás a két osztály között
- ez általában az jelzi, hogy a felelősségeket rosszul osztottuk ki
- refaktorálásként azt tehetjük, hogy a megfelelő attribútumokat és metódusokat átrendezzük a két osztály között, ezáltal növeljük az osztályok belső kohézióját, és csökkentjük a közöttük lévő csatolás mértékét
- lehetőség szerint azt mindenképpen el kellene érni, hogy a két osztály között ne legyen körkörös függőség
- ha a kliens a protected tagokhoz akar hozzáférni, akkor érdemes elgondolkozni azon, hogy delegálás helyett inkább öröklést használjunk

Alternative classes with different interfaces

- *"Ugyanarra a feladatra különböző interfészű osztályok"*
- például, ha többfajta operációs rendszer kell támogatnunk, és az operációs rendszer funkcióit meghívó osztályok eltérő publikus interfésszel rendelkeznek
    - ilyenkor ezek nem használhatók egységesen, a klienskód sokkal bonyolultabbá válik
- refaktorálásként átnevezhetjük az egyes függvényeket, hogy közeledjen egymáshoz a két osztály interfésze
- próbáljunk meg valamilyen közös ősosztályt, vagy közös interfészt találni, amely segítségével a kliens egységesen tudja kezelni az operációs rendszer specifikus osztályokat

- a cél tehát az, hogy valamilyen módon közös interfészre jussunk
- ötleteket meríthetünk a Bridge tervezési mintából is

Incomplete library class

- *"Egy felhasznált könyvtárbeli osztályt nem tudunk módosítani"*
- tehát egy felhasznált könyvtárbeli osztályban nincsen meg minden funkció, amire nekünk szükségünk van, de annak a működését nem tudjuk átírni, így refaktorálásra sincsen lehetőség
- egy lehetséges megoldás az lehet, hogy a könyvtárbeli osztályt becsomagoljuk egy saját osztályba, és delegálás segítségével adunk hozzá új funkcionalitást
    - ez a megoldás azért is előnyös lehet, mert hogyha később egy másik könyvtárra kell átállnunk, akkor csak a delegáló függvényt kell átírni, a kód többi részét nem.
- amennyiben a könyvtár lehetőséget ad rá, a szerver osztály funkcionalitását öröklődéssel is ki tudjuk bővíteni
- egyes programnyelvekben másféle kibővítési lehetőségek is lehetnek, nagyon hasznos lehet pl C#-ban az extension metódusok használata

Data class

- *"Egy osztály csak adatot tárol"*
- egy ilyen osztály általában annak az eredménye, hogy nem objektum orientáltan terveztünk
- az osztály nem rendelkezik felelősségekkel, a viselkedés általában más osztályok, tipikusan egy isten-osztály végzi
- refaktorálásként tehát azt tehetjük, hogy a kliensosztályokból metódusok átmozgatásával vigyünk át felelősségeket az adattároló osztályba
- elképzelhető az is, hogy egy kliensmetódusnak csak egy részét kell átmozgatnunk
- érdemes azt is megfontolni, hogy refaktorálásként töröljük a setter metódusokat, vagy azok nagy részét, és az állapotot ne egy külső osztály manipulálja, hanem az adatosztály által nyert felelősségeket implementáló függvények

Refused bequest (elutasított örökség)

- *"Egy leszármazottnak nincsen szüksége az ős viselkedésére"*
- ennek oka általában az, hogy az ősben definiált viselkedés túl magasra került az öröklési hierarchiában
    - a refaktorálási lépés az lehet, hogy ezt a függvényt és a hozzá tarozó attribútumokat toljuk lejjebb az öröklési hierarchiában
- ugyanennek a code smell-nek egy tipikus este lehet az is, ha a leszármazott osztály egy üres függvénnyel definiálja felül az ősben megadott viselkedést
    - ez nyilvánvalóan a Liskov-féle helyettesíthetőségi elvnek a megsértése, és azt jelzi, hogy az osztályok nem megfelelő sorrendben szerepelnek az öröklési hierarchiában
    - refaktorálásként ilyenkor azt tehetjük, hogy átrendezzük az öröklési hierarchiát, például felcseréljük sz őst a leszármazottal
- egy harmadik lehetséges jele lehet ennek a code smellnek az, ha a leszármazottban az ős interfésze egyáltalán nem releváns

    - ez általában annak a jele, hogy az öröklést nem a viselkedés újrahasznosításaként használjuk
    - a refaktorálás ilyenkor az lehet, hogy az öröklődést delegálással helyettesítjük

Comments

- *"Túl sok magyarázó komment van a kódban"*
- vannak jó és vannak rossz fajta kommentek
- a rossz kommentek közé tartoznak azok, melyek a kód működését próbálják megmagyarázni
- a probléma ilyenkor az, hogy a kód túlságosan bonyolult, és a kommentek próbálnak ezen valamilyen szinten enyhíteni, általában nem sok sikerrel
- a cél az lenne, hogy tiszta kódot írjunk, amelyet könnyű olvasni, ne pedig egy bonyolult kódot próbáljuk meg kommentekkel kozmetikázni
- refaktorálásként tehát azt tehetjük, hogy a bonyolult kódot feldaraboljuk kisebb függvényekre, a függvényeknek és a változóknak pedig értelmes nevet adunk
    - a clean code ebben fog majd segíteni
- a kommentekben tehát nem azt kell leírni, hogy mit csinál a kód, hanem azt, hogy miért csinálja azt, amit
- a kommentekben a tervezői döntéseket kell dokumentálnunk
- a kódnak pedig olyan olvashatónak kell lennie, hogy mindig kiderüljön egyértelműen, hogy mit csinál

Downcasting

- *"Típuskasztolás és típusellenőrzés a kódban"*
- ezek az esetek azt jelzik, hogy a konkrét típustól függ a kód, és nem használjuk ki a polimorfizmus nyújtotta lehetőségeket
- ilyenkor sérül az OCP is, hiszen, ha később új típusok jönnek be, akkor valószínűleg majd rájuk is kell típustesztelést alkalmazni, vagyis át kell írni a meglévő kódot
- ha valahol típusellenőrzést találunk, akkor célszerű úgy refaktorálnunk, hogy kihasználjuk a polimorfizmus nyújtotta lehetőségeket
- a típusellenőrzés a TDA elv megsértését is jelezheti, ilyenkor refaktorálásként a kliens kódból vigyük át a funkcionalitást a szerverbe

Refaktorálási technikák

- a refaktorálási technikák önmagukban nem jók vagy rosszak, csak egy eszközkészletet adnak arra, hogy a kódban milyen transzformációkat tudunk elvégezni
- éppen ezért egyes transzformációknál az inverz is szerepelni fog, mert mindig az adott implementációtól és code smell-től függ, hogy melyiket kell közülük használni

Extract method

- fogjuk a kód egy részét és függvényt készítünk belőle
- ezzel például csökkenthető a kódduplikáció

Inline method

- a függvény törzsét beépítjük a hívó oldalon

- ez például akkor lehet hasznos, ha egy felesleges delegációt kell megszűntetni

Inline temporary

- ha egy ideiglenes változót csak egyszer használunk, akkor megszabadulhatunk tőle úgy, ha az értékét behelyettesítjük a használat helyén

Replace temporary with query

- egy ideiglenes változó helyett függvényt vezet be

Introduce explaining variable

- egy bonyolultabb kifejezést helyettesít egy olvashatóbb nevű változóval

Split temporary variable

- akkor lehet hasznos, ha egy ideiglenes változót több különböző célra is felhasználunk
- ekkor válasszuk szét ezeket a célokat, és adjunk különböző nevet a változóknak

Remove assignments to parameters

- ennek során egy ideiglenes változót vezetünk be ahelyett, hogy a paraméternek adnánk értéket

Replace method with method object

- ennek során egy metódusból osztályt készítünk, ahol a lokális változókból mezők lesznek
- ez tipikusan arra való, hogy a függvényben definiált algoritmust kiszervezzük egy külső osztályba

Substitute algorithm

- célja, hogy az algoritmust lecseréljük egy tisztább változatra

Move method

- ennek során egy metódust egy másik osztályba mozgatunk át

Move field

- ennek során egy mezőt egy másik osztályba mozgatunk át

Extract class

- fogjuk az osztály néhány metódusát és mezőjét, és ezekből egy másik osztályt képzünk

Inline class

- egy osztály metódusait és mezőit beépítjük egy másik osztályba

Hide delegate

- egy távoli objektumhoz intézett hívást delegációval helyettesítünk

Remove middle man

- a delegációt egy direkt hívásra cseréljük le

Introduce foreign method

- a kliens osztályban egy új metódust készítünk, ahol a szerver a paraméter
- ez például arra használható, hogy a kliensben olyan funkciót adjunk a szerverhez, amelyet a szerver alapból nem támogat

Introduce local extension

- úgy bővíti ki a szerver működését, hogy egy új leszármazottat hoz belőle létre

Self encapsulate field

- ennek során egy mezőt getter-setter metódusokba, vagy property-be csomagolunk

Replace data value with object

- egyszerű primitív adatokat tároló változók helyett egy osztályt vezet be

Change value to reference

- értékegyenlőség helyett referencia-egyenlőséget vezet be
- ilyen például a Flyweight tervezési minta

Change reference to value

- referencia-egyenlőség helyett értékegyenlőséget vezet be

Replace array with object

- arra való, hogy ha egy tömböt használunk arra, hogy összetartozó elemeket csoportosítsunk, akkor ezt a tömböt cseréljük le egy osztályra, ahol az osztály mezői a tömb egyes elemei lesznek

Duplicate observed data

- a GUI-ban ne tároljunk olyan adatokat, amelyek a modell számára fontosak, helyette válasszuk szét a modell és a GUI réteget
- a GUI függjön a modelltől, és ne fordítva

Change unidirectional association to bidirectional

- az asszociáció átjárhatóságát változtatja meg
- egyirányú asszociációt kétirányúra vált

Change bidirectional association to unidirectional

- az asszociáció átjárhatóságát változtatja meg
- kétirányú asszociációt egyirányúra vált

Replace magic numbers with symbolic constant

- a kódban szétszórt számok és string literálok helyett nevesített konstansokat használjunk

Encapsulate field

- nem privát attribútumból privátot készítünk
- az értékéhez való hozzáférést getter-setter metódusokkal, vagy property-vel oldjuk meg

Encapsulate collection

- egy gyűjtemény típusú attribútumot gyűjtünk el
- kifelé ennek csak egy olvasható nézetét publikáljuk ki
- a módosítására pedig saját függvényeket publikálunk

Replace record with data class

- akkor lehet hasznos, ha egy hagyományos, nem oo nyelven írt könyvtárhoz kell csatlakoznunk, és ilyenkor a rekordok reprezentálására adatosztályokat használunk

Replace type code with class

- célja, hogy a típusok egész számként vagy enum-ként való kódolása helyett különböző osztályokat használjunk

Replace type code with subclasses

- célja, hogy a típusok egész számként vagy enum-ként való kódolása helyett különböző osztályokat használjunk
- és ezen túlmenően öröklődést és polimorfizmust is használ

Replace type code with state/strategy

- célja, hogy a típusok egész számként vagy enum-ként való kódolása helyett különböző osztályokat használjunk
- ha az öröklődés valamilyen ok miatt nem működik, akkor ennek segítségével az objektum dinamikus viselkedését szervezzük ki a state vagy a strategy tervezési minta segítségével

Replace subclass with fields

- akkor hasznos, ha a leszármazottak nem adnak hozzá viselkedést az őshöz
- ilyenkor az egyes leszármazottak csak állapotban különböznek, helyettük az ős is példányosítható lenne különböző kezdőállapotokkal
- felesleges a leszármazottakat külön típusként definiálni

Decompose conditional

- ennek során a feltételes elágazások különböző ágait önálló függvényekként reprezentáljuk

Consolidate conditional expression

- azt jelenti, hogy ha ugyanaz a feltétel többször egymás után szerepel, akkor a kódot írjuk át úgy, hogy azt csak egyszer kelljen kiértékelni, és egyszer kelljen rá tesztelni

Consolidate duplicate conditional fragments

- azt jelenti, hogy ha ugyanaz a részkifejezés többször egymás után szerepel, akkor a kódot írjuk át úgy, hogy azt csak egyszer kelljen kiértékelni, és egyszer kelljen rá tesztelni

Remove control flag

- egy kilépést jelző változót cseréljünk le egy break vagy return utasításra

Replace nested conditional with guard clauses

- célja, hogy az egymásba ágyazott feltételek helyett if-ek, vagy if-else-k sorozatát kapjuk
- vagyis egy lapos feltételsorozatot szeretnénk elérni

Replace conditional with polymorphism

- a típus-ellenőrzéseket polimorfizmussal helyettesítjük

Introduce null object

- célja, hogy ne null értékeket tároljunk, hanem használjuk a null object tervezési mintát, vagyis olyan objektumokat, amelyek a default működést produkálják
- így nem kell folyton a null értékekre rátesztelni

Introduce assertion

- a kód egy adott pontján feltételezett állapotra egy feltételellenőrzést helyezünk el, amely debug módban kivételt dob, ha a feltétel sérül, így fejlesztés közben ellenőrizhető, hogy mindig a megfelelő állapotban hívjuk az objektumot

Rename method

- ennek során a függvényt átnevezzük úgy, hogy az új név jobban tükrözze a függvény célját

Add parameter

- új paramétert adunk a függvényhez, hogy több információt tudjunk átadni neki

Remove parameter

- egy nem használt paramétert törlünk a függvény fejlécéből

Separate query from modifier

- ennek során két metódust készítünk, az egyik csak lekérdezésre, a másik csak módosításra való

Parameterize method

- célja, hogy két nagyon hasonló függvényt kombinálunk össze eggyé, néhány újabb paraméter bevezetésével

Replace parameter with explicit methods

- célja egy metódus felosztása több darab, kevesebb paraméterrel rendelkező metódusra

Preserve whole object

- ezzel a transzformációval a teljes objektumra adjuk tovább referenciát, nem pedig az objektum egyes részeit adjuk tovább paraméterként

Replace parameter with method

- nem egy metódushívás eredményét adjuk tovább paraméterként, hanem hagyjuk, hogy ezt a hívást a függvény saját maga végezze el

Introduce parameter object

- a sok csoportosuló paramétert helyettesíti egy önálló osztállyal

Remove setting method

- ennek során az attribútumok értékét a konstruktorokban állítjuk be, settereket pedig nem biztosítunk hozzájuk

Hide method

- ha egy osztályból egy publikus függvényt senki sem használ, akkor tegyük ezt a függvényt priváttá
- feleslegesen ne szemeteljük tele a publikus interfészt

Replace constructor with factory method

- akkor lehet hasznos, ha egy objektum létrehozása és inicializálása bonyolultabb annál, mint egy egyszerű konstruktor-hívás

Encapsulate downcast

- azt jelenti, hogy ha egy metódus eredményét a klienseknek folyton át kell kasztolniuk, akkor ezt a típuskasztolást tegyük át a metódusba, és a metódus térjen vissza a helyes típussal

Replace error code with exception

- azt jelenti, hogy hibakódok, és speciális visszatérési értékek helyett használjunk kivételeket

Replace exception with test

- azt jelenti, hogy kivételek elkapása helyett ellenőrizzük le, hogy a beadott paraméterek helyesek-e, vagyis teljesítjük-e a hívott függvény előfeltételeit

Pull up field

- ennek során a leszármazottakban lévő közös attribútomokat a közös ősbe visszük át

Pull up method

- ennek során a leszármazottakban lévő közös metódusokat a közös ősbe visszük át

Pull up constructor body

- ennek során a leszármazottak konstruktorainak közös részeit viszik fel a közös ős konstruktorába

Push down method

- egy olyan metódust, amely a leszármazottak csak egy részére érvényes, lejjebb viszünk az öröklési hierarchiában

Push down field

- egy olyan mezőt, amely a leszármazottak csak egy részére érvényes, lejjebb viszünk az öröklési hierarchiában

Extract subclass

- ennek során azokat a viselkedéseket, amelyek csak a leszármazottak egy részére érvényesek egy önálló leszármazottba szervezzük ki, és az eredeti leszármazottak tőle fognak a továbbiakban leszármazni

Extract superclass

- ha néhány leszármazott közös viselkedéssel rendelkezik, akkor ezt a viselkedést egy önálló közös ősbe lehet kiszervezni

Extract interface

- azt jelenti, hogy ha néhány osztály publikus interfészében vannak közös részek, akkor az a közös halmaz egy közös interfészbe kiszervezhető

Collapse hierarchy

- ha egy osztály nem sok mindent ad hozzá az ős viselkedéséhez, akkor összevonjuk őt az őssel

Form template method

- azt jelenti, hogy ha néhány leszármazott hasonló metódusokat tartalmaz hasonló lépésekkel azonos sorrendben, akkor a template method tervezési mintát alkalmazva vigyük fel őket az ősbe

Replace inheritance with delegation

- azt jelenti, hogy ha egy leszármazottnak nem mindenre van szüksége abból, amit az őstől megörököl, akkor öröklődés helyett használjunk delegációt

Replace delegation with inheritance

- akkor használjuk, ha egy osztály többnyire csak delegál egy hasonló interfészű másik osztályhoz, akkor a delegálást kiválthatjuk egy örökléssel

Tease apart inheritance

- azt jelenti, hogy ha egy öröklési hierarchia két dolgot csinál egyszerre, akkor vágjuk szét két különálló öröklési hierarchiára

Convert procedural design to objects

- ennek során az adatosztályokhoz felelősségeket, tehát metódusokat rendelünk

Separate domain from presentation

- azt jelenti, hogy az üzleti logikát vegyük ki a GUI-ból és hozzunk létre önálló, csak az üzleti logikával foglalkozó osztályokat

Extract hierarchy

- egy bonyolult osztály darabolunk fel egy komplett öröklési hierarchiára
- a cél az, hogy a speciális eseteket az egyes leszármazottak kezeljék

Clean code

Clean code fogalma

Rossz kód

- a kódot akkor nevezzük rossznak, ha nehéz olvasni, nehéz megérteni és nehéz karbantartani
- rossz kód keletkezhet olyankor,
    - ha nagyon sietünk, és úgy gondoljuk, nincs időnk rendesen megcsinálni a dolgot
    - ha már nagyon elegünk van a szoftverből, és csak túl akarunk lenni az egészen
    - vagy keletkezhet lustaságból, tapasztalatlanságból, vagy halogatásból, hogy majd később úgyis rendbe rakjuk (és ez a később általában azt jelenti, hogy soha)
- a rossz kód azért hátrányos, mert csökkenti a produktivitást
- ilyenkor egyes menedzserek azt gondolják, hogy ha még több embert raknak a projektre, akkor növelhető a produktivitás, de általában ezzel csak rontanak a helyzeten, mert az újabb emberek még nem értik a kódot, nem látják át teljesen, nem ismerik az eredeti tervezési döntéseket, és ráadásul szűk határidő alatti nyomásban kell dolgozniuk
    - ennek következtében a helyzet csak rosszabbodik, és később csak még nehezebb lesz hozzányúlni a rendszerhez
- fontos azonban megjegyezni, hogy a rossz kódért nem a menedzser a felelős, hanem a fejlesztő
    - a fejlesztő az, aki írja a kódot, és attól még, hogy rövid a határidő, lehet szép kódot készíteni
- az egyetlen módja annak, hogy hosszútávon is produktívak legyünk, hogy a kódok mindig, minden helyzetben szépen és tisztán kell tartani

Clean code (tiszta kód)

- a clean code azért fontos, mert a kódot többet olvassuk, mint írjuk
- az arány körülbelül 10 az 1-hez szokott lenni
- azt, hogy mi a tiszta kód, nagyon nehéz egyértelműen definiálni
- néhány általános szabály adható, ezeket fogjuk hamarosan megtanulni
- de a clean code inkább művészet, mint egzakt tudomány
- az ember általában felismeri, de ez még nem azt jelenti, hogy képes is tiszta kódot írni
- sok gyakorlásra és tapasztalatra van szükség, hogy odáig eljusson valaki

Clean code tulajdonságai

- könnyű olvasni
- könnyű megérteni
- elegánsan fejezi ki magát
- hatékony
- kevés függőséggel rendelkezik más komponensek felé
- minden lehetséges hibát lekezel
- egyetlen dologra fókuszál, és azt jól csinálja

- könnyen továbbfejleszthető

Kifejező nevek írása

"Az osztálynevek főnév jellegűek legyenek!"

- kerüljük az olyan általános neveket, mint a Manager, Data, Processzor vagy Info az osztálynevekben
- használjunk minél specifikusabb nevet
- az osztály neve lehetőleg ne legyen ige, kivéve akkor, ha a viselkedést szándékosan kiszervezzük, mint például a Visitor vagy Strategy tervezési minta esetén

"A függvénynevek ige jellegűek legyenek"

- mivel a függvények valamilyen tevékenységet hajtanak végre
- ez tipikusan azt jelenti, hogy a függvény neve egy ige, például save, vagy valamilyen főnév-ige kombináció, például deletePage
- a getter setter függvények neveit Javaban konvenció szerint get, set vagy is igével szoktuk kezdeni
- amennyiben egy osztálynak overload-olt konstruktorai vannak, érdemes ezeket priváttá tenni, és helyettük factory metódusokat publikálni, amelyek a nevükben tudnak utalni arra, hogy milyen objektumot szeretnénk létrehozni

"Használjuk az alkalmazási terület elnevezéseit"

- tehát ha egy biológusnak írunk szoftvert, akkor a biológus által használt elnevezéseket, ha pedig egy építésznek, akkor az építészek által használt elnevezéseket használjuk
- ez azért lehet előnyös, mert ha változnak a követelmények, akkor könnyebb megtalálni azt a helyet a kódban, ahol javítani kell
- az informatikus szakma egyik érdekes kihívása az, hogy bele kell ásnunk magunkat abba a területbe, meg kell tanulnunk azt a nyelvet, amit a megrendelő beszél

"Használjunk beszédes neveket!"

- az alábbi kóddal az a probléma, hogy a nevek nem kifejezőek
  - a függvény neve nem utal arra, hogy mit csinál a függvény, a lista nevéből nem derül ki, hogy mit tárolunk benne, és az sem derül ki, hogy miért olyan fontos a 0 index, illetve, hogy mit jelent a 4-es szám
  - a d változónak pedig adhatnánk olyan nevet, hogy a kommentre ne legyen szükség

```
public List<int[]> getThem() {
      List<int[]> list1= new List<int[]>();
      for (int[] xin theList)
            if (x[0] == 4)
                  list1.add(x);
      return list1;
}
int d; // elapsed time in days
```

- nem mindig egyszerű ilyet találni, de érdemes rászánni az időt, mert később nagyon sok időt megspórolhatunk vele

- egy név akkor beszédes, ha elmondja, hogy az adott dolog miért létezik, mit csinál és hogyan lehet használni
- ha egy név kommentezésre szorul, akkor az a név rossz, válasszunk helyette mást
- nyugodtan használhatunk hosszabb neveket is, hiszen a modern fejlesztőkörnyezetek segítenek a nevek kiegészítésében
- a példa egy tisztább megoldása az alábbi lehet:

```
public List<int[]> getFlaggedCells() {
      List<int[]> flaggedCells= new List<int[]>();
      for (int[] cellin gameBoard)
            if (cell[Consts.StatusValue] == Consts.Flagged)
                  flaggedCells.add(cell);
      return flaggedCells;
}
intelapsedTimeInDays;
```

"Kerüljük a félreinformálást!"

- ne használjunk félrevezető neveket, és olyanokat, melyeket nehéz megkülönböztetni egymástól
- a megoldás tehát az, hogy használjunk minél rövidebb, de lényegre törő neveket
- ne nevezzünk List-nek egy nem lista objektumot
- ne használjunk olyan neveket, melyek csak nagyon kicsit különböznek egymástól
- ne használjunk O és l betűket változók elnevezésére, mert ezeket egyes betűtípusokban nehéz megkülönböztetni a 0-tól és az 1-től

"Használjunk megkülönböztető neveket"

- kerüljük a számozott neveket, illetve a zaj jellegű elő és utótagokat, például Info, Data, The stb...
- egy függvényen belül pedig ne használjuk fel ugyanazt a változót több, különböző célból
- a fordító elég okos ahhoz, hogy ha egy regisztert már nem használunk, akkor azt egy másik változóhoz rendeli hozzá
- a problémánk megoldása tehát az, hogy a paraméterek nevéből derüljön ki, hogy melyik mire való
- a függvények nevei is utaljanak arra, hogy melyik függvény mit csinál pontosan

"Használjunk kimondható neveket!"

- ellenkező esetben nagyon nehéz lesz a többi fejlesztővel kommunikálni
- ne használjunk csak mássalhangzókból álló rövidítéseket

"Használjunk kereshető neveket!"

- ezeket könnyebb megjegyezni, és később rájuk keresni
- éppen ezért fontos, hogy a kódba ne égessünk bele számokat, illetve string literálokat, hanem használjunk helyettük kereshető, nevesíthető konstansokat
- ugyancsak kerüljük a rövidítéseket, mert ezek is rontják a kereshetőséget
- érdemes a betűszavak használatát is kerülni
- ha azonban a betűszavak szabványosak, és jól ismertek, akkor mindenképpen ezeket használjuk, és ne írjuk ki a teljes nevet
    - ilyen betűszavak a GUI, XML és HTML is

"Kerüljük a névkódolást!"

- kerülendő például a hungarian notation, ahol a változó típusa rövidítve szerepel a változó nevében
    - a probléma ezzel a megoldással az, hogy ha a változó típusa megváltozik, akkor a változót át kell nevezni
    - ezt elég egyszer elfelejteni, és máris inkonzisztencia lép fel
    - a modern fordítók és az erőse típusos nyelvek úgyis segítenek a típusok helyes kezelésében
- a szabály alapján ugyancsak ne használjunk prefixeket az egyes változókra
    - például a tagváltozók nevét ne kezdjük m_ -al
    - ez csak kényelmetlenné teszi a fejlesztést, hiszen minden változó előtt ezt a két karaktert le kell majd ütni
    - a modern fejlesztő környezetek ki tudják színezni különböző színűre a lokális változókat és a tagváltozókat
- a típusok elnevezésében is érdemes kerülni a prefixeket és a suffixeket
    - a Java API nagyon szépen tartja ezt a szabályt
    - például a listát definiáló interfészt List-nek hívják, az implementáló osztályt pedig ArrayList-nek vagy LinkedList-nek
    - ezzel ellentétben .NET-ben IList-nek hívák az interfészt, és List-nek az implementáló osztályt
    - a .NET-es konvenció sérti a szabályunkat
- sok esetben azonban Java-ban sem lehet elkerülni a szabály megsértését
    - ha például nem tudunk jobb nevet találni egy interfészt implementáló osztálynak, akkor az implementációból származó impl szócskát tegyük az implementáció végére
    - sok esetben tehát nem lehet ezt a szabályt tartani, hiszen nem hívhatjuk ugyanolyan néven az interfészt és az ő egyetlen implementációját biztosító osztályt
- természetesen ne ragaszkodjunk ehhez a szabályhoz akkor, ha a használt nyelv vagy keretrendszer konvenciói eltérnek ettől
    - például a konvenció .NET-ben az, hogy minden interfészt nagy I betűvel kezdünk
    - ezt a konvenciót mindenképpen tartsuk meg .NET-ben különben össze fogjuk zavarni a többi fejlesztőt
- ugyancsak eltérhetünk a szabálytól, ha a céges kódolási szabvány ellent mond neki
    - érdemes azonban erre a szabályra gondolni, amikor egy céges kódolási szabványt hozunk létre

"Használunk a hatókörnek megfelelő hosszúságú nevet!"

- ha függvény és paramétereinek neve túl rövidek, akkor nem derül ki, hogy mire valóak
- ha a függvény belsejében a ciklusváltozó túlságosan hosszú, akkor ciklus olvashatatlanná válik
- amiket kívülről sokan látnak és sokan használnak (például függvénynevek), azok legyenek hosszabbak
- azok a nevek, amelyek csak egy rövid kódblokkban érdekesek (mint például a ciklusváltozók, vagy a lambda paraméterek), azok nyugodtan lehetnek rövidek

- az általános szabály a clean code szerint az, hogy a rövid nevek jobbak a hosszabb neveknél, de csak abban az esetben, ha elég kifejezőek
- ha nem sikerül rövid, kifejező nevet találni, akkor érdemes elmenni a hosszabb nevek irányába
- nem kell félni a hosszabb nevektől sem, hiszen a fejlesztőkörnyezetek úgyis segítenek a nevek kiegészítéséében
- de ha lehet, akkor találjunk egy rövidebb, és kifejezőbb nevet

"Adott dologra mindig ugyanazt a nevet használjuk!"

- a dolgok lekérdezésére használjuk mindig ugyanazt az igét, ne keverjük a Fetch, Retrieve és Get igéket a kódunkban, mert az inkonzisztenciához vezet
- a cél az, hogy az elnevezések mindig konzisztensek legyenek
- az inkonzisztenciával az a baj, hogy összezavarja a fejlesztőt, ráadásul nagyon nehéz megtanulni, és emlékezni rá, hogy mikor melyik változatot kell használni

"Biztosítsunk értelmes kontextust a változóknak!"

- nem minden névből következik ugyanis egyértelműen, hogy pontosan mit jelent
- az angol state szó egy címen belül államot jelent, egy állapotgépen belül pedig egy állapotot
- fontos tehát az, hogy a kontextus biztosítson elég információt ahhoz, hogy a név jelentése mindig egyértelmű legyen

"Kerüljük az értelmetlen kontextust!"

- például felesleges a cég nevét minden osztály előtt prefixként szerepeltetni, mert az nem ad semmit hozzá a név jelentéséhez
- ráadásul a fejlesztőknek nagyon nehéz dolguk lesz a fejlesztőkörnyezetekben, mert csak a prefix beírása után fognak segítséget kapni a név kiegészítéséhez

Függvényekkel kapcsolatos clean code szabályok

"A függvények legyenek rövidek"

- A függvények legyenek rövidek, lehetőleg 2-4 sor hosszúak, de semmiképpen se legyen 20 sornál hosszabbak
- hiszen, ha egy képernyőre sem férnek rá, akkor nagyon nehéz átlátni a működésüket

"A blokkok belseje egyetlen sor legyen!"

- ez vonatkozik a feltételes elágazásokra, a ciklusokra, a try-catch blokkokra és a többi hasonló megoldásra
- ha egynél több sorra lenne szükség, akkor szervezzük ki őket egy külön függvénybe, és tarthatjuk az egy sort azáltal, hogy azt a függvényt hívjuk meg abban az egy sorban
- hogyha ezt betartjuk, annak dokumentációs ereje is van, hiszen azt a függvényt majd el kell neveznünk valahogyan
- a szabály egyik következménye az is, hogy blokkok nem lehetnek egymásba ágyazva
- a beágyazott blokkokat is ki kell szervezni külön függvénybe
- ez egyben azt is jelenti, hogy a függvények belsejében a bekezdések mélysége legfeljebb 1 vagy 2 lehet

- ennek köszönhetően pedig sokkal egyszerűbb megérteni a függvények működését

"Egy függvény egyszerre csak egy dolgot csináljon!"

- ez az egy dolog éppen egy szinttel van a függvény neve által jelzett absztrakciós szint alatt
- fontos figyelembe venni, hogy a kivételkezelés, a naplózás és konkurenciakezelés mind önálló absztrakciós szintek, önállóan egy-egy feladatok
- ha tehát egy függvény ezek egyikével már foglalkozik, akkor mást már nem csinálhat, az egyéb feladatokat egy általa hívott másik függvénynek kell végrehajtania

"Egy függvény egyszerre csak egy absztrakciós szinten dolgozzon!"

- egy függvényen belül tehát minden utasítás ugyanahhoz az absztrakciós szinthez tartozzon
- egy függvényen belül ne keverjük az absztrakciós szinteket, mert akkor az olvasó nehezen fogja tudni eldönteni, hogy egy adott kifejezés számára fontos, vagy éppen irreleváns
- azért sem jó, ha keverednek az absztrakciós szintek, mert az ilyen feladatok csak vonzzák és vonzzák az újabb és újabb feladatokat
- egy idő után azt vesszük észre, hogy a függvény túlságosan naggyá és bonyolulttá válik

"A függvények lépcsőzetesen, az absztrakciós szinteket fokozatosan kibontva kövessék egymást!"

- ez a szabály azt jelenti, hogy ha egy függvény hív más függvényeket, akkor ezek a függvények közvetlenül a hívó után szerepelnek a kódban
- hogyha ez betartjuk, akkor a kódot tényleg úgy lehet majd olvasni, mintha egy próza lenne
- előbb kapunk egy magasabb absztrakciós szintű áttekintő képet, majd leássuk magunkat a mélyebb és mélyebb magyarázatokba
- ez a szabály segít abban is, hogy a függvényeken belül konzisztensen tudjuk tartani az absztrakciós szinteket

"Egy függvénynek minél kevesebb paramétere legyen!"

- ideális esetben ez a szám 0, de az 1 illetve 2 paraméter is még elfogadható
    - a 3 paraméter kerülendő
    - 3-nál több paramétere pedig semmi esetre se legyen a függvénynek
- ha sok a paraméter, akkor az általában azt jelzi, hogy keverednek az absztrakciós szintek a függvényen belül
- a sok paraméter azért is hátrányos, mert tesztelésnél sok értékkombináció is lehet, így nagyon nehéz minden esetre letesztelni a függvényt
- az ideális eset tehát az, ha a függvénynek nincsen paramétere
- az 1 paraméteres függvényt két dolog indokolhatja:
    - vagy a paraméterről kérdezünk valamit a függvény segítségével,
    - vagy valamilyen műveletet végzünk a paraméteren, és a transzformált értékkel térünk vissza
- a 2 paraméteres függvényeket már nehezebb megérteni, mint az 1 paramétereseket
    - 2 paraméternek akkor lehet értelme, ha a paramétereknek van egy természetes sorrendje

    - ilyenek például az x, y koordináták
    - ha ilyen természetes sorrend nem létezik, akkor a függvény neve utaljon arra, hogy melyik paraméter mit jelent (lényegében a függvény nevébe belekódolhatjuk a paraméterek jelentését)
- a 3 paraméteres függvényeket még nehezebb megérteni, még nehezebb tesztelni, így érdemes őket elkerülni
- ha 3-nál több paraméterünk van, akkor érdemes elgondolkodni a refaktorálásnál tanul paraméterobjektum bevezetésén
    - a paramétereket ekkor egyetlen osztályba gyűjtjük össze, a függvény így egyetlen paraméteressé válik, a paraméter típusa pedig ez az osztály

"Kerüljük a mellékhatásokat!"

- egy függvény azt ígéri, hogy egyszerre csak egy dolgot csinál
- a mellékhatás egy olyan rejtett dolog, melyre a hívó nem számít
    - ilyen mellékhatás lehet egy osztály mezőjének átállítása, a beérkező paraméterek állapotának megváltoztatása, vagy éppen globális változók átállítása
- a mellékhatásokkal az is a baj, hogy időbeli csatolásokat hoznak létre, vagyis a függvények csak adott időpontokban, vagy csak adott sorrendben hívhatóak meg
- a mellékhatások konkurenciaproblémákhoz is vezethetnek, vagyis nehéz lesz több szálon használni az alkalmazást

"Válasszuk szét a parancs és a lekérdezés jellegű függvényeket!"

- egy függvény vagy csináljon valamit, vagy válaszoljon egy kérdésre, de ne egyszerre a kettőt
- a szabály alól kivételt jelentenek az atomi műveletként végrehajtandó szálbiztos műveletek (ezek általában a test&set jellegű műveletek)

Hogyan írjunk függvényeket?

- először általában egyszerűbb megírni a függvényeket hosszabban, sok paraméterrel, sok elágazással
- majd ezután refaktorálási lépésekkel fel lehet darabolni egyszerűbb, és jobban érthető függvényekre
- nagyon sok gyakorlás kell ahhoz, hogy az ember rutinszerűen be tudja tartani az itt tanult szabályokat

Kommentek írása és elhagyása

Kommentek használata

- a kommentek arra valók, hogy kifejezzünk olyasmit, amit kódban nem tudunk kifejezni
- a jól elhelyezett kommentek nagyon hasznosak tudnak lenni, de az értelmetlen és akár félrevezető kommentek nagyon károsak lehetnek
    - ezeknél még az is jobb lenne, ha nem is szerepelnének a kódban
- érdemes végig gondolni, hogy az egyetlen hiteles információforrás maga a programkód
    - ha ez a kód csúnya, és nehezen értelmezhető, akkor ne kommentet írjunk hozzá, hanem írjuk át, hogy olvasható legyen

- a clean code elvek szerint tehát nem kommentekkel kell dokumentálni a kódot, hanem a cél az, hogy a programkód a saját olvashatósága révén dokumentálja saját magát
- éppen ezért, nagyon sok helyen feleslegesek a kommentek, de van néhány eset, ahol indokolt a használatuk

Motyogás

- rossz kommentnek számít a motyogás, vagyis amikor nem derül ki a komment pontos jelentése
- vagy ha az adott kontextusban a komment nem értelmezhető, és így arra kényszerülünk, hogy a kód más részeit is megvizsgáljuk, vagy éppen valamilyen dokumentációban kelljen utánanézni bizonyos elemeknek

Redundáns komment, Félrevezető komment

- egy ilyen komment csak megismétli azt, ami a kódból is kiolvasható, így túl sok hozzáadott értéke nincs
- ha ráadásul nem elég precíz, akkor még félrevezető is lehet

Kötelező kommentek, Zaj jellegű kommentek

- ugyancsak károsak a kötelező kommentek, melyeknek nincs semmilyen hozzáadott értékük
- ilyen kommentek tipikusan akkor keletkeznek, ha előírás mindennek a kommentelése, és így a programozó csak a fejlesztőkörnyezetre hagyja, hogy generáljon néhány alapértelmezett kommentet
- ezek a kommentek csak tele szemetelik a kódot, és nehezen olvashatóvá teszik
- éppen ezért ne tegyük előírássá azt, hogy minden egyes függvényt, minden egyes változót kommentezni kelljen
- ha mégis előírjuk, akkor annak az lesz a következménye, hogy egy csomó zaj jellegű komment keletkezik
- egy konstruktorról mindenki látja, hogy ha az egy default konstruktor, a változó neve pedig utal arra, hogy mit tárolunk benne
- ennél már csak az rosszabb, hogy ha egy programozó úgy hoz létre új változót, hogy egy korábbit a kommentjével együtt lemásol, majd a zaj jellegű kommentet elfelejti frissíteni

Ideges komment

- az idegességből hagyott kommentek nem csak zajt okoznak, hanem érdemi információtartalmuk sincsen, így teljesen feleslegesek
- ha már valakinek annyi energiája van, hogy idegeskedjen, ekkor ezt az energiát úgy is le lehet vezetni, hogy átírja szebbé a kódot

Komment függvény vagy változó helyett

- ugyancsak kerülendő a komment akkor, ha a kód átírásával a komment feleslegessé válik
- ne írjunk kommentet akkor, ha ugyanazt az információt kódban is ki tudjuk fejezni
- írjunk a komment helyett inkább kifejező kódot
- érthetőbbé tehető a kód például úgy is, ha a részkifejezéseket értelmes nevű változókban tároljuk el

Banner jellegű kommentek

- kerüljük a feltűnő, banner jellegű kommenteket, amelyek csak az arcunkba villognak
- ezek zajosak, és csak elterelik a figyelmet a fontosabb kommentekről
- pl.: //////// Banner komment ///////////

Blokkzáró kommentek

- kerüljük a blokkzáró kommenteket is, mert hosszú függvények esetén nehéz őket konzisztensen tartani
- egyébként se írjunk hosszú függvényeket, és akkor nincs szükség arra, hogy kommentekkel jelöljük, hogy melyik blokkot zárjuk éppen

Kikommentezett kód

- rossz kódnak számít a kikommentezett kód is
- a kikommentezett kóddal az a baj, hogy fontosnak tűnik
- lehet, hogy csak emlékeztető céljuk van, lehet, hogy valamilyen közelgő változásra hívják fel a figyelmet, de az is lehet, hogy valaki évekkel ezelőtt ott felejtette őket
- más fejlesztőknek nincs meg a bátorságuk, hogy kitöröljék őket, és így csak folyamatosan gyűlnek az évek során
- ne hagyjunk tehát kikommentezett kódot a forráskódban, használjuk a verziókezelő rendszert, amelyből bármikor kibányászható a kód korábbi változata

Napló jellegű kommentek

- ugyancsak kerülendőek a napló jellegű kommentek, vagyis, hogy a kódot ki, mikor, milyen célból módosította
- használjunk helyette verziókezelő rendszert, melynek feladata éppen ezeknek az információknak a rendszerezett tárolása

Szerzőt hirdető kommentek

- ezek helyett is célravezetőbb a verziókezelő rendszer használata

HTML kommentek

- érdemes kerülni a HTML-ben formázott kommenteket is
- ezek csak a kigenerált HTML dokumentációban lesznek olvashatók, a kódban nagyon nehéz őket értelmezni
- pedig a kommenteknek többnyire a programkódban van értelme
- a HTML markerek hozzáadásának felelőssége, a dokumentáció-generátor eszközre tartozik

Nemlokális információ kommentben

- rossz kommentnek számít az is, ha a kommenten belül egy nem lokális információ szerepel
- nemlokális az az információ, amely a rendszer egy másik moduljában, vagy valami nagyon távoli helyen van specifikálva
- a kommentben inkább a specifikálás helyét kellene meghivatkozni

Túl sok információt tartalmazó komment

- kerüljük a túl sok irreleváns információt is

- például, ha a függvényünk a dokumentumot XML-é konvertálja, akkor kommentként ne az XML szabványt másoljuk be
- akkor van értelme bekommentezni az XML szabvány egyes pontjait, ha magát az XML szabvány implementáló könyvtárat készítünk
- egyedül itt lehet értelme hangsúlyozni bizonyos pontokat a szabványból
- de egy egyszerű XML konverziónál ne a szabványt kezdjük el idézni

Hiányzó kapcsolat kommentben

- rosszak azok a kommentek is, melyeknek a kóddal való kapcsolatuk hiányos
- ha például a komment és a kód között hiányzik a konzisztencia
- fontos tehát, hogy a komment és a kód közötti kapcsolat nyilvánvaló legyen
- a komment célja, hogy segítse a kód megértését
- ha már maga a komment is magyarázatot igényel, akkor az a komment rossz

Informatív kommentek

- ezek tényleges, hozzáadott értékekkel rendelkeznek, olyan információkat szolgáltatnak, amelyek a kódból nem olvashatóak ki
- sokszor azonban még ezek is elhagyhatók, ha sikerül egy találóbb nevet választani
- például reguláris kifejezést tartalmazó kód kommentezése hasznos lehet, mert azt könnyebb értelmezni, mint magát a reguláris kifejezést

Szándék, illetve tervezői döntés magyarázatát tartalmazó komment

- ezt érdemes használni, sőt, azt is mondhatjuk, hogy ez a legfontosabb indok arra, hogy kommentet hagyjunk a kódban

Tisztázó kommentek

- ugyancsak hasznosak lehetnek azok a kommentek, melyek valamilyen bonyolultabb szintaktikát tartalmazó kódot tesznek olvashatóbbá
- ezek azonban nagyon kockázatosak, mert módosítások esetén elcsúszhatnak a kód valódi jelentésétől
- bánjunk tehát velük nagyon óvatosan, és ne felejtsük el őket szinkronizálni a kóddal

Figyelmeztetés a következményre a kommentben

- hasznosak azok a kommentek is, amelyek valamilyen veszélyre figyelmeztetnek
- például ilyen az, ha megjelöljük kommentben, hogy egy osztály nem szálbiztos, vagy hogy egy teszt futtatása nagyon sok ideig tart

TODO kommentek

- valamilyen elvégzendő feladatra hívják fel a figyelmet
- ezekre könnyű rákeresni, de a jobb fejlesztőkörnyezetek akár ki is gyűjtik ezeket nekünk

Hangsúlyozó kommentek

- ugyancsak fontosak azok a kommentek, melyek valamit hangsúlyoznak
- például a bug fix során bekerült részeket érdemes kommentben jelölni, és megjegyezni, hogy ezeket a hatékonyság érdekében ne töröljük ki

Publikus API dokumentálása

- ugyancsak jó és követendő, a publikus API dokumentálása
- amikor egy olyan könyvtárat készítünk, melyet mások használni fognak, akkor fontos, hogy annak publikus API-ja jól dokumentált legyen
- a könyvtár implementálása során érdemes a clean code elveket követni, így belül nem szükségek dokumentációs kommentekkel ellátni a nem publikus dolgokat, azonban a kívülről látható publikus API-t érdemes dokumentációs kommentekkel ellátni
- erre azért van szükség, mert a könyvtár használói nem fogják böngészni a könyvtár forráskódját
- az egyetlen hivatalos információforrásuk a könyvtár publikus API-jának dokumentációja lesz
- fontos az is, hogy a dokumentációs kommenteket ne csak úgy generáltassuk a fejlesztőkörnyezettel, hanem szánjunk rá időt, és írjunk bele minden szükséges információt

Publikus API dokumentálásának tartalma

- mit csinál az adott osztály vagy függvény
- melyik paraméter mit jelent
- mi a paraméterek értékkészlete
- mik a függvények előfeltételei is és utófeltételei
- mik az osztály invariánsai
- milyen esetekben milyen kivételeket dobálhat egy függvény
- melyik függvény szálbiztos, és melyik nem
- érdemes mellékelni példakódot, hogy hogyan lehet meghívni a könyvtár egyes függvényeit
- célszerű azt is leírni, hogy a függvény által megvalósított algoritmusnak milyen a hatékonysága

Kivételek definiálása és kezelése

"Hibakódok helyett használjunk kivételeket!"

- a kivételek segítenek abban, hogy az alkalmazás logikáját elválasszuk a hibakezelést megvalósító logikától
- korábban már beszéltünk arról, hogy egy függvény egyszerre csak egy dologgal foglalkozzon
- egy dolognak számít az alkalmazás logikája, és egy másik dolognak számít a kivételkezelés
- ez azt is jelenti, hogy a try és a catch blokkok belsejét külön fájlba szervezzük ki
- fontos az is, hogy a catch blokkok csak hibakezelést tartalmazzanak, ne legyen bennük alkalmazáslogika
- az alkalmazáslogika a try blokkokba való

"Biztosítsunk kontextust a kivételhez!"

- példa: egy függvény minden hiba esetén egy általános kivételt dob
    - a probléma ezzel a megoldással az, hogy nem lehet tudni a hiba okát, és nem lehet finomhangoltan reagálni rá

- legyen benne a kivételben az az információ, hogy melyik fájlban és hol keletkezett, mi volt a hiba oka, és hogyan tudná elkerülni a programozó, hogy ilyen kivételt kapjon
- célszerű a kivételben annyi információt biztosítani, amely naplózható, és a napló alapján rekonstruálható a probléma
- a kivételek üzenetei mindig a programozóknak szólnak, sohasem a végfelhasználóknak
- éppen ezért a kivétel szövegében a programozónak kell biztosítani olyan információt, amely segítségével elkerülheti, hogy később ilyen hibaüzenetet kapjon

"Használjunk uchecked (runtime) kivételeket!"

- a checked exception-nek az a hátránya, hogy a kliensre rákényszerítjük a hiba kezelését, még akkor is, ha nem tudja, mit kezdjen vele
- a checked exception-ök használata azért sem jó ötlet, mert könnyű velük megsérteni a Liskov-féle helyettesíthetőség elvét
- tegyük fel ugyanis, hogy van egy interfészünk, amelyet implementálunk egy osztályban, az osztály implementációjában azonban olyan függvényt hívunk, amely olyan checked kivételt generálhat, amelyet az osztály nem tud lekezelni, így tovább szeretné dobni, azonban ezt nem teheti meg, mert az implementált interfész függvényeinek szignatúrájából hiányzik ez a checked exception
- ha viszont ezt a checked exception-t végig vezetjük az öröklési hierarchián, akkor az open-closed principle elvet sértjük meg
- többek között ezek az indokok vannak amögött is, hogy a C# nyelvben nincsenek cheked exception-ök
- és éppen ezért Javaban is érdemes a runtime, vagyis unchecked exception-ökre hagyatkozni

"A kivételeket mindig a hívó szemszögéből definiáljuk!"

- ne azt nézzük, hogy nálunk hányféle módon történhet hiba, hanem hogy a hívó oldal milyen értelmes módon tud ezekre reagálni
- például, ha egy kliens üzenetet szeretne elküldeni a hálózaton, és az nem sikerül, akkor a klienst általában csak a sikertelenség ténye érdekli
- számára irreleváns az, hogy az alsó protokollban melyik bit romlott el
- ne kényszerítsük a klienst arra, hogy sokfajta catch blokkban kelljen kezelnie a kivételeket
- ha egy külső könyvtár erre készítene minket, akkor az ő kivételeit csomagoljuk be egy egyszerűbb kivételbe
- általánosan is igaz, hogy egy külső könyvtárhoz érdemes saját csomagolóosztályokat készíteni
- ez minimalizálja a tőle való függést, könnyebbé tesz egy esetleges átállást egy másik könyvtárra, könnyebb mockolni a könyvtárat fejlesztés céljából, és nem kötnek minket a könyvtár által választott tervezői döntések
- így olyan API-t definiálhatunk magunknak, ami számunkra kényelmes

"Ne térjünk vissza null értékkel!"

- ha valami rosszul sült el, akkor dobjunk kivételt, ha pedig nincs mivel visszatérni, akkor használjuk a Null object tervezési mintát
- egy kollekció esetén a null object az üres kollekció

- fontos tehát, hogy azok a függvények, melyek visszatérési értéke egy kollekció, sose nullértékkel térjenek vissza, hanem egy üres kollekcióval
- így a kliensкódban az eredmény mindig bejárható, nem kell a kliensnek null-ellenőrzésekkel bajlódnia
- a megoldás az, hogy a beérkező paraméterek előfeltételeit minél előbb ellenőrizzük, többek között azt is, hogy az értékük nem null-e, és mielőbb jelezzük a hívónak, ha valamilyen előfeltétel sérült
    - egyrészt így a belső ellenőrzésekben a null feltételek teljesen elhagyhatók, másrészt garantáltan nem fogunk nagyon mélyről nullpointer exception-öket kapni
    - ha a függvények nem térnek vissza nullértékkel, akkor a kollekciók is egyből bejárhatók, mindenféle nullellenőrzés nélkül

"Ne adjunk át paraméterként null értéket!"

- kivéve akkor, ha az API explicit megengedi
- különben nagyon mélyről fogunk nullpointer exception-öket kapni

Bejövő paraméterek ellenőrzése

- amikor egy osztályban egy publikus függvényt készítünk, akkor érdemes ellenőrizni, hogy a bejövő paraméterek értéke megfelelő-e
- ha valami nem stimmel, akkor egy kivétellel rögtön jelezzük a hívó számára, hogy rossz a bejövő paraméter
- a kivételben jelezzük azt is, hogy mi lenne az elfogadható értékkészlet
- a privát függvények már feltételezhetik, hogy a bejövő paraméterek már helyesek, így azokban már nem kell ellenőrizgetni, hogy valóban jók-e az értékek

Objektumok és adatstruktúrák

Procedurális fejlesztés vs objektumorientált fejlesztés

Procedurális kód	Objektumorientált kód
a fókuszban az adat áll	a fókuszban a viselkedés áll
a viselkedést külön függvényekben implementáljuk, amelyek paraméterként kapják meg az adatokat	az adatreprezentáció el van rejtve
könnyű új függvényt hozzáadni anélkül, hogy változtatnánk az adatstruktúrán	könnyű megváltoztatni a belső adatreprezentációt anélkül, hogy a külső interfész változna
nehéz megváltoztatni az adatstruktúrát, mert az minden függvényre hatással lehet	nehéz megváltoztatni a publikus interfészt, mert az egész öröklési hierarchiára, és minden hívóra kihathat

stabil, ritkán változó adatstruktúrát kíván, ilyen például az adatbázis táblákra való leképzés (ORM), valamint a backend és a frontend közötti kommunikációt megvalósító DT (az ilyen stabil adatstruktúráknál a Demeter-törvény is megsérthető, hiszen, ha az adatstruktúra stabil, akkor a hosszú hívási láncok is stabilak)	stabil viselkedést kíván, a működés kibővítését leszármazással és delegációval valósítjuk meg (erről szól az OCP, vagyis meglévő osztályok kódjához, viselkedéséhez nem nyúlunk hozzá, zártak vagyunk a módosításra, de nyitottak a bővítésre, vagyis leszármazással és delegációval újabb viselkedést adhatunk a rendszerhez)

Procedurális fejlesztés vs objektumorientált fejlesztés választása

- ahogy a legtöbb paradigmánál, itt sem lehet azt mondani, hogy az egyik vagy a másik módszer jobb
- mindig a követelményektől és a körülményektől függ, hogy melyiket érdemes választani
- amit az objektumorientált stílusban nehéz megcsinálni, azt a procedurálisban könnyű, és fordítva
- sőt, egy komplex alkalmazás estén keveredhetnek is a stílusok
- például az adatbázisok és a hálózati kommunikáció esetén az adatok dominálnak
- az üzleti logikát érdemes objektumorientált stílusban fejleszteni

API tervezési elvek

API

- Application Programming Interface
- azon függvények és osztályok gyűjteménye, melyet egy általunk felhasznált könyvtár publikál
- ha fejlesztők vagyunk, egyben API tervezők is vagyunk, hiszen a moduljainkat más fejlesztők is felhasználják
- éppen ezért érdemes végig gondolni, hogy mások számára milyen lehetőségeket biztosítunk
- ha mások elkezdik használni az általunk publikált modult, akkor azt már sokkal nehezebb lesz később megváltoztatni
- fontos tehát, hogy egy jól átgondolt, stabil interfészt adjunk ki a külvilágnak
- egy interfész megtervezésében segítenek az API tervezési elvek

Jó API tulajdonságai

Könnyű megtanulni, és memorizálni

- ebben segíthet az, hogyha az API kicsi, egyszerű kevés dolgot kell megtanulni hozzá
- nagy segítség az is, ha az API konzisztens, tehát ugyanazok a fogalmak ugyanolyan néven, különböző fogalmak különböző néven szerepelnek az API-ban
- konzisztens a függvényparaméterek sorrendje, konzisztens az, hogy mit milyen sorrendben kell hívni, és ha az API hasonlít egy másik API-ra, akkor ahhoz is konzisztensen tartja magát

Olvasható kódhoz vezet

- ehhez az kell, hogy az API által biztosított függvények a megfelelő absztrakciós szinten legyenek
- ne rejtsenek el fontos információkat az API-t hívó kliensektől, és ne is kényszerítsék a klienseket arra, hogy számukra irreleváns információkat kezeljenek

Nehéz rosszul használni

- könnyebb benne helyes kódot írni, mint rosszat
- egy jó API nem köti meg a felhasználó kezét azzal, hogy a propery-ket csak adott sorrendben lehet beállítani, vagy függvényeket csak adott sorrendben lehet meghívni
- egy jó API-nak nincsenek olyan mellékhatásai, melyek kihatással lehetnének a kliensekre

Könnyű kiterjeszteni

- egyrészt magának az API-nak a könyvtára is fejlődik idővel, bejönnek új osztályok, új függvények, új paraméterek, új enum érékek, és az API-t már eleve úgy kell megtervezni, hogy ezek később ne okozzanak gondot
- a kiterjeszthetőség a kliensek számára is fontos, hiszen lehetnek olyan esetek, melyet az API könyvtára nem fed le, de a klienseknek mégis szükségük van rájuk, és meg kell

adni a klienseknek a lehetőséget arra, hogy ezt a plusz funkciót hozzáadhassák a könyvtárhoz
- ehhez azonban az kell, hogy az API biztosítson kibővítési pontokat öröklődéssel, delegációcal, dependency injection-nel, vagy egyéb kiterjesztési lehetőségekkel

Teljes, vagyis lefedi az összes felhasználói igényt

- ez természetes az ideális eset, a gyakorlatban ritkán valósul meg
- éppen ezért fontos az, hogy az API biztosítson kiterjesztési lehetőségeket

Jól dokumentált

- ha az API-hoz nincsen dokumentáció, a kliensek nem tudják, hogyan kell használni
- minden egyes dokumentáló elemre, például osztályra, függvényre meg kel mondani, hogy az micsoda, mit csinál, melyik paraméter mit jelent, mi a paraméterek lehetséges értékkészlete, mik az elő- és utófeltételek, mik az invariánsok, milyen estekben milyen kivételek keletkezhetnek, mely függvények szálbiztosak, milyen az algoritmusok hatékonysága
- és mindenképpen tartalmazzon a dokumentáció példakódokat, vagyis, hogy hogyan lehet használni az API-t

API fejlesztés folyamata

API fejlesztés lépései

1. Gyűjtsük össze a követelményeket
2. Írjunk use-case-eket
3. Vegyünk példát hasonló API megoldásokról
4. Definiáljuk az API-t
5. Ellenőriztessük másokkal
6. Írjunk sok-sok példát
7. Készüljünk fel a kiterjesztésekre
8. Implementáljuk
9. Ha kételkedünk, hagyjuk ki
10. Ne változtassunk rajta

1. Gyűjtsük össze a követelményeket

- ez néha egyszerű, ha egy szabványt kell implementálni
- de néha nagyon nehéz is tud lenni, ha a követelmények nem tiszták
- éppen ezért érdemes minél több embert megkérdezni, hogy ők miket tartanak fontosnak
- ezek az emberek lehetnek kollégák, a főnökünk, de leginkább azokat érdemes megkérdezni, akik majd használni fogják az API-t
- vigyázzunk arra, hogy néha követelmények helyett megoldásokat fogunk kapni, ezekkel bánjunk óvatosan, mert lehet, hogy létezik jobb megoldás is

2. Írjunk use-case-eket

- vagyis, hogy milyen használati eseteket kell az API-nak lefednie

- fontos, hogy ez a lépés előzze meg az implementációt, és az API a felhasználók tényleges igényét tükrözze, ne pedig az implementáció belső struktúráját
- érdemes egyszerű mintakódokat is írni az egyes use-case-ekhez, hogy lássuk milyen interfészt várunk el az API-tól
- egyelőre ne törődjünk azzal, hogy ezeket milyen nehéz lesz majd implementálni, a cél az, hogy könnyű legyen majd használni az API-t
- ezek a kis minta kódok a tesztelésnél és a dokumentáció elkészítésénél is hasznosak lesznek

3. Vegyünk példát hasonló API megoldásokról

- ha van már egy évek óta használt, jól bevált stabil API, akkor érdemes annak a mintáját követnünk
- egy ilyen API éveken át fejlődött, sok-sok felhasználói visszajelzéssel, nem érdemes nekünk valami teljesen újat kitalálni
- azért is jó, hogyha hasonlítunk egy meglévő API-ra, mert akkor a mi API-nkat is könnyű lesz megtanulni, a másik API-val ismerős felhasználók könnyen kiismerik majd magukat a mi általunk fejlesztett API-ban is
- természetesen, ha a másik API kényelmetlen, netán éppen emiatt írunk egy újat, akkor nem kell követnünk a másik API által meghatározott szabályokat
- figyeljünk azért arra, hogy egy API nem biztos, hogy teljesen rossz, lehetnek benne jó ötletek, melyeket érdemes átvenni
- ha a célunk az, hogy egy rossz API-t kiváltsunk egy általunk fejlesztett újabbal, akkor fontoljuk meg azt, hogy a régebbi API-hoz képest definiálunk újabb, egyszerűbb dolgokat, de egy ideig visszafele kompatibilisek maradunk, amíg a teljes átállást sikerül megvalósítani

4. Definiáljuk az API-t

- a követelmények és a use-case-ek birtokában definiáljuk az API-t, vagyis adjuk meg a pontos osztályokat és függvényeket, amelyeket majd használni lehet
- törekedjünk arra, hogy az API kívülről könnyen használható legyen, még akkor is, ha az implementációja majd trükkös lesz
- írjunk a use-case-ek alapján unit teszteket, ezek segíthetnek abban, hogy az esetleges hiányosságokat feltárjuk
- később majd jók lesznek arra is, hogy az API implementációját tesztelhessük velük
- ez a lépés abban is segíthet, hogy az implementációs részletek ne szivárogjanak fel az API felületére
- esetleg néhány beállítást kivezethetünk az API felületére, amelyek segíthetnek a hatékonyság finomhangolásában, de azért ezekkel is óvatosan bánjunk

5. Ellenőriztessük másokkal

- ellenőriztessük az API definícióját főnökünkkel, kollégáinkkal, és a potenciális felhasználókkal
- gyűjtsünk minél több visszajelzést, legyenek azok pozitívak vagy negatívak, minden vélemény nagyon hasznos tud lenni
- ehhez azonban fontos az, hogy az API definíciója ne legyen túlságosan hosszú, lehetőleg férjen rá egy A4-es oldalra
- így az emberek gyorsan áttekinthetik, és nekünk is egyszerűbb karbantartani azt

6. Írjunk sok-sok példát

- ha megkaptuk a visszajelzéseket, és néhány iterációs lépés után sikerült kitalálni a végleges API definíciót, akkor kezdjünk el sok-sok példát írni, hogy hogyan is lehet majd használni az API-t
- kiindulásként felhasználhatjuk a use-case-eket, és ezek a példák segíthetnek abban is, hogy az API definícióját a többiek segítségével véglegesítsük
- ahogy az API fejlődik, mindig frissítsük ezeket a példákat
- és fontos az is, hogy mielőbb kezdjük el használni az API-t a saját kódjainkban, mert nincs annál jobb módszer a hiányosságok kiküszöbölésére, mint hogy a saját főztünket esszük

7. Készüljünk fel a kiterjesztésekre

- az API az idők során fejlődni fog
- egyrészt mi magunk, a fejlesztők is fogunk hozzáadni újabb elemeket az API-nkhoz, másrészt a felhasználóknak is meg kell adni a lehetőséget, hogy kiterjesszék a működését
- a kiterjesztés működhet öröklődéssel, delegációval, dependency injection-nel, vagy egy service provider interfész implementálásával, és az implementáció beregisztrálásával
- ahhoz, hogy fel tudjunk készülni a megfelelő kiterjeszthetőségre, minden kiterjesztési lehetőségre legalább 3 példát írjunk saját magunk is
- ezekből látszani fog, hogy az ősben definiált virtuális függvények elegendőek-e ahhoz, hogy a kiterjesztések széles skáláját biztosíthassák

8. Implementáljuk

- ebbe beletartozik az is, hogy jó alaposan teszteljük le az API működését
- figyeljünk arra is, hogy néha úgy keletkezik publikus API, hogy egy belső API-t teszünk nyilvánossá
- ilyenkor mindenképpen érdemes átnézni az API-t, nehogy valamit hibásan publikáljunk, mert később aztán azt nagyon nehéz lesz javítani

9. Ha kételkedünk, hagyjuk ki

- mielőtt publikálnánk az API-t, meg egyszer alaposan vizsgáljuk meg
- ha bármilyen funkcionalitásban is kételkedünk, inkább hagyjuk ki, vagy váltsuk át belső API-ra, de bizonytalan dolgot ne adjunk ki a kezünk közül
- hozzáadni mindig tudunk az API-hoz elvenni belőle azonban sosem
- érdemes béta változatokat kiadni, és figyelni a felhasználói visszajelzéseket, de merjünk nem-et mondani, mert nem járhatunk mindenkinek a kedvében
- készüljünk fel arra is, hogy vétünk majd hibákat, de azokat majd az évek során ki fogjuk küszöbölni, az API folyamatosan fejlődni fog

10. Ne változtassunk rajta

- ha publikáltuk az API-t, már ne változtassunk rajta
- fontos, hogy megőrizzük a visszafele kompatibilitást
- jobbá tehetjük a dokumentációt, megváltoztathatjuk a belső implementációt, behozhatunk újabb feature-öket, de amit egyszer az API-ból kipublikáltunk, azon már ne változtassunk

- ha így tennénk, azzal egy csomó másik ember kódját rontanánk el
- valószínűleg nagyon megharagudnának ránk, és nem használnák többet a könyvtárunkat
- nagyon nagy tehát rajtunk a felelősség, mert az API-t elsőre jól el kell találnunk
- egy kipublikált könyvtárat verziózni is kell, és érdemes követni a szemantikus verziózást

Szemantikus verzió

- a szemantikus verzió általában 3 darab számból áll: major, a minor és a patch verziószámból
- a major, vagyis a fő verziószámot akkor kell növelni, ha az API-ban egy visszafele inkompatibilis változtatást hajtottunk végre
    - ennek nagyon ritkán szabadna csak megtörténnie
- a minor, vagyis az alverzió-számot akkor növeljük, hogyha új verziót adunk az API-hoz, de az API továbbra is visszafele kompatibilis marad
- az utolsó, patch verziószám akkor növekszik, hogyha a könyvtár API-ján nem változtatunk, csak a belső implementációban javítottunk valamilyen hibát

API tervezési elvek

"Válasszunk magától értetődő neveket és szignatúrákat!"

- érdemes itt is a clean code-ban tanult elnevezési szabályokat követni
- ami még fontos, hogy a paraméterek sorrendje mindig konzisztens legyen

"Válasszunk egyfajta nevet az összetartozó dolgokra"

- például a grafikus komponenseknél ne keverjük a control és a widget elnevezéseket
- ha viszont vannak hasonló dolgok, amiket meg kell különböztetni egymástól, azok kapjanak különböző neveket

"Kerüljük a hamis konzisztenciát"

- ez azt jelenti, hogy ha már valamire van konvenció, akkor annak a konvenciónak az elnevezéseit ne használjuk másra
- például Java-ban a setter függvények set kulcsszóval kezdődnek, ne használjunk hasonló nevű függvényeket más célra

"Kerüljük a rövidítéseket"

- a rövidítésekkel az a baj, hogy nem nyilvánvalóak, nehéz őket megtanulni
- kivételt képeznek a szabály alól a jól bevált és jól ismert rövidítések
    - ilyen például a min, illetve a max, a minimum és a maximum helyett
- a jól ismert betűszavakra sem vonatkozik a szabály
    - például használjuk az XML rövidítést, az Extensible Markup Language kiírása helyett

"Általános nevek helyett használjunk specifikus neveket"

- ezek jobban rámutatnak arra, hogy az elnevezett dolognak mi is a feladata

- később még mindig lehet általánosítani, ha szükséges, de ha egy általános nevet elhasználunk, akkor egy más, de hasonló célra már nehezebb lesz egy másik általános nevet választani

"Használjuk a helyi terminológiát"

- ismerjük meg a célplatform és célprogramozási környezet elnevezéseit, és konvencióit, és használjuk ezt a terminológiát a mi API-nkban is
- ez vonatkozik arra is, ha az API-nkat portolni szeretnénk egy másik környezetbe
- például a Java nyelvű API-ban definiált getter setter-eket a C# nyelvű API-ban property-kre cseréljük le
- az új környezetben definiált API felhasználói boldogok lesznek, hiszen nem térünk el az általuk használt konvencióktól, a meglévő környezet felhasználóit pedig nem fogja zavarni, mert ők úgyis maradnak a saját környezetüknél

"Ne legyünk az alattunk lévő API elnevezésinek foglyai"

- ha tehát egy másik API-ra építünk, amely rosszul megválasztott elnevezéseket használ, nyugodtan definiáljuk a saját terminológiánkat

"Válasszunk megfelelő alapértelmezett értékeket és működést"

- érdemes az alapértelmezett viselkedéssel a használati esetek nagy részét lefedni
- így a felhasználók egyből használhatják az API-t, nem kell a konfigurációval bajlódniuk
- természetesen a dokumentációban tűntessük fel az alapértelmezett értékeket
- a boolean típusú értékeket érdemes úgy elnevezni, hogy az alapértelmezett érték a hamis legyen
- például egy grafikus komponens láthatóságát jelölő property neve ne visible, vagyis látható legyen, hanem hidden, vagyis rejtett

"Az API-t ne tegyük túlságosan okossá"

- ez azt jelenti, hogy ne legyenek olyan rejtett mellékhatások, amelyekre a kliens nem számít
- például, ha egy komponens szövegét beállítjuk valamire, akkor a setter ne próbálja meg felismerni, hogy ennek a szövegnek a formátuma html, és ne állítsa át a szöveg formátumot html-re
- a felhasználó mindösszesen csak a komponens szövegét szerette volna átírni, és nagyon meg fog lepődni, ha ennek hatására egy másik property értéke is átáll

"Gondoljunk az API tervezési döntéseink teljesítménybeli következményeire"

- például, ha egy típus nem csak olvasható, akkor védelmi másolatokat kell majd belőle készíteni, többszálúság esetén pedig kölcsönös kizárással kell majd védeni
- fontos az is, hogy az API-t ne forgassuk ki azért, hogy teljesítményt nyerjünk vele
- általában a jó terv egybeesik a jó teljesítménnyel is

"Figyeljünk a szélső esetekre"

- általában ezekre a szélső esetekre épül a többi eset
- fontos az is, hogy a szélső esteket ne kelljen speciálisan kezelni
- például, ha egy keresésnek nincs eredménye, akkor ne null értékkel térjünk vissza, hanem egy üres kollekcióval

- így a kliens kódnak nem kell folyton a null értékeket ellenőrizgetnie
- tesztellésnél térjünk ki a szélső értékek külön tesztelésére is

"Minimalizáljuk a módosíthatóságot"

- használjunk tehát immutable, vagyis csak olvasható osztályokat
- ezek egyszerűek, szálbiztosak, és beadhatók paraméterként, vagy visszaadhatók eredményként anélkül, hogy aggódnunk kellene, hogy valaki tönkre teszi a belső állapotukat
- a módosítható osztályokat védeni kell védelmi másolatokkal, a többszálú esetben pedig kölcsönös kizárással
- ezek mind rontják az alkalmazás teljesítményét

"Vagy öröklésre tervezzünk, és dokumentáljuk is, vagy tiltsuk meg az öröklődést"

- érdemes az a konzervatív megoldást követni, hogy minden osztály legyen alapból final vagy sealed, tehát ne lehessen belőle leszármazni
- és később csak azokat nyissuk ki, amelyeket kifejezettek kifejtési pontként szeretnénk definiálni az API-ban

"Legyünk óvatosak, ha virtuális API-t definiálunk"

- nagyon nehéz a kiterjesztési pontként definiált virtuális ősosztály granualitását pontosan eltalálni
- ha túl kevés a virtuálissá tett függvény, nem lehet elég erős kiterjesztéseket írni
- ha viszont túl sok függvényt teszünk virtuálissá, néhány metódust nagyon veszélyes lesz felüldefiniálni
- a szabály az, hogy a publikus függvények ne legyenek virtuálisak, és a templete method tervezési minta segítségével legyenek implementálva, így meghívva a virtuálissá tett függvényeket, amelyek láthatósága viszont legyen protected

"Egy GUI-hoz készülő API tulajdonságokra épüljön"

- Ha grafikus komponenseket készítünk, azoknak az API tulajdonságokra, vagyis property-kre épüljön
- a grafikus komponenseknél tipikusan nagyon sok mindent lehet állítgatni
- ne kényszerítsük a felhasználókat arra, hogy mindent a konstruktorban kelljen beadniuk paraméterként
- válasszunk jó alapértelmezett értékeket, és amennyiben szükséges, a felhasználók a konstuktorhívás után átállíthatják a property-k értékét
- figyeljünk arra is, hogy ne kössük meg azt, hogy a property-ket milyen sorrendben kell beállítani, tetszőleges sorrend legyen elfogadható

"Próbáljuk meg előre látni a testreszabhatósági lehetőségeket"

- biztosítsuk a felhasználóknak azt, hogy property-ken, és egyéb kiterjesztési lehetőségeken keresztül tudják finom hangolni a könyvtárunk viselkedését
- de vigyázzunk arra is, hogy limitáljuk a lehetséges opciók számát
- csak olyan opciókat publikáljunk, ami a legtöbb felhasználó számára érdekes lehet
- jobb, ha egy egyszerűbb API-nk van kevesebb opcióval, mint egy bonyolult API nagyon sok beállítási lehetőséggel

"Kerüljük a hosszú paraméterlistákat"

- 3 vagy kevesebb paraméter az ideális, ha ennél több van, akkor a felhasználó általában már kénytelen a dokumentációt is megnézni
- ha túl sok egyforma típusú paraméterünk van, akkor a felhasználó könnyen hibázhat, és összekeverheti őket
- ha egy függvényünknek mégis túl sok paramétere lenne, akkor vagy daraboljuk fel azt a függvényt, vagy cseréljük le a hosszú paraméter listát egy paraméter objektumra

"Használjunk kényelmi függvényeket"

- ez azt jelenti, hogy a gyakran igénybe vett feladatokra legyenek olyan függvények, amelyek ezeket egyből elvégzik, és ne kelljen a felhasználónak sok-sok függvényhívást intéznie egy-egy egyszerűbb feladat végrehajtásához
- ebben a tekintetben például a .NET API sokkal kényelmesebb, mint a Java API
- érdemes arra figyelni, hogy amikor a saját API-nkat implementáljuk, számos olyan kényelmi és utility függvényt készítsünk, amely akár másoknak is hasznos lehet
- vizsgáljuk meg, hogy ezek közül miket lehetne az API külső interfészén publikálni

"Induljunk el 3 sorból"

- ez a 3 sor csak szimbolikus, valójában a 3 sor 3 lépést takar
- az első lépés az inicializálás
    - tipikusan valamilyen konstruktorhívás
- a második lépés az alapvető konfiguráció
    - a property értékek beállítása, amennyiben el szeretnénk térni az alapértelmezett működéstől
- a harmadik lépés a futtatás
    - vagyis az API elindítása
- nyilvánvalóan ez a 3 lépés lehet kevesebb, mint 3 sor, vagy több, mint 3 sor, de a lényeg, hogy ezek a fő lépések
- ne térjünk el ettől, és ne kényszerítsük a felhasználót arra, hogy bonyolult inicializáló kódrészekkel kelljen berúgnia az API-t
- ellenkező esetben ezek a bonyolult kódrészek a StackOverflow-n keresztül fognak elkezdeni terjedni, mint valami vírus

"A mágia rendben van, a számok nem"

- az API belső implementációjánál lehet, hogy trükközni kell, de az API külső interfészét mindenképpen tartsuk tisztán
- speciális értékek helyett használjunk nevesített konstansokat, vagy enum értékeket

"Jelezzük a hibát, amilyen hamar csak lehet"

- a legjobb, ha már fordítási időben kiderül, ha valami nem stimmel
- az API-ban deklarált függvények mindig ellenőrizzék a saját prekondíciójukat, és ha valamelyik előfeltétel sérül, azonnal jelezzék azt egy kivétel formájában
- fontos az is, hogy ne nyeljük el a hibákat, ha valami nem stimmel, azt mindenképpen jelezzük kivétellel
- vigyázzunk azonban arra, hogy kivételekkel tényleg csak a hibákat jelöljük, a kivételeket ne használjuk az alkalmazás-logika vezérlésére

"Ne használjunk checked exception-öket"

- a probléma velük az, hogy sértik az OCP elvet és a Liskov-féle helyettesítési elvet is
- további probléma velük az, hogy a klienst felesleges kódírásra kényszerítik

"A kivétel jelezze, hogyan lehet őt elkerülni"

- a kivétel tehát ne csak egy utalást tartalmazzon a problémára, hanem a kivétel szövegében szerepeljen az is, milyen lépéseket tehet a felhasználó, hogy többé ne kapja ezt a kivételt
- ez tipikusan az szokott lenni, hogy mi a függvény prekondíciója, például, hogy egy bemenő paraméter értéke nem lehet null, és ilyenkor a felhasználó írhat egy ellenőrzést, hogy nehogy véletlenül null értéket adjon be paraméterként
- a kivétel jelentheti az is, hogy valamilyen konfigurációs beállítás hiányzik, és célszerűen ilyenkor a kivétel szövege adja meg azt, hogy melyik fájlban, konkrétan melyik helyen, mi az a beállítás, ami hiányzik
- ne a dokumentációból kelljen a fejlesztőnek előbogarásznia, hogy mik a pontos beállítások, és ne valamilyen általános hiba alapján kelljen az interneten keresgélnie, hogy mit is rontott el pontosan

"Vigyázzunk a függvények túlterhelésével"

- sokszor érdemes különböző függvénynevet adni a különböző változatoknak
- így a függvény neve jobban utalhat arra, hogy a függvény neve mire is utalhat pontosan
- hogyha mégis a túlterhelés mellett döntünk, akkor vigyázzunk arra, hogy mindig egyértelmű legyen a működés, a kevesebb paraméteres overload azt csinálja, mintha a több paraméteres overload-ot hívnánk alapértelmezett paraméterekkel

"Teszteljük ki a belét is"

- írjunk unit teszteket, írjuk regressziós teszteket, használjuk fel a use-case-eket, és azt a sok-sok példát, amiket tervezésnél megalkottunk, írjunk teszteket a szélső értékekre
- a lényeg az, hogy nagyon alaposan teszteljük ki az API-t
- senki nem szeret egy bug-os könyvtárra építeni, így hamar ott hagyják a könyvtárunkat, ha nem tudnak vele dolgozni

"Dokumentáljuk az API-t"

- dokumentáció nélküli könyvtárat nagyon nehéz, szinte lehetetlen használni
- sajnos a legtöbb nyílt forráskódú könyvtár ettől szenved
- ha a felhasználó nem tud a dokumentációra támaszkodni, akkor hamar ott fogja hagyni a könyvtárat

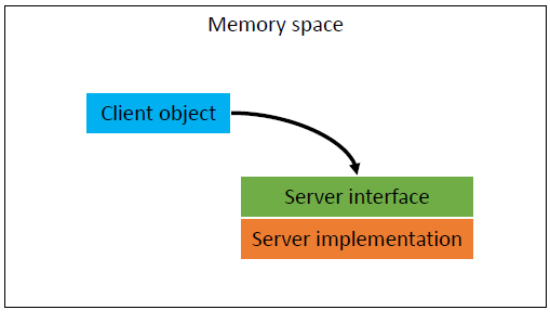
Elosztott OO

Elosztott OO

- ebben a témakörben azt vizsgáljuk meg, hogy ha a hívó, vagyis a kliens objektum és a hívott, vagyis a szerver objektum hálózati kapcsolaton kommunikál egymással, akkor ennek során milyen problémák léphetnek fel, és milyen megoldásokat adhatunk ezekre a problémákra

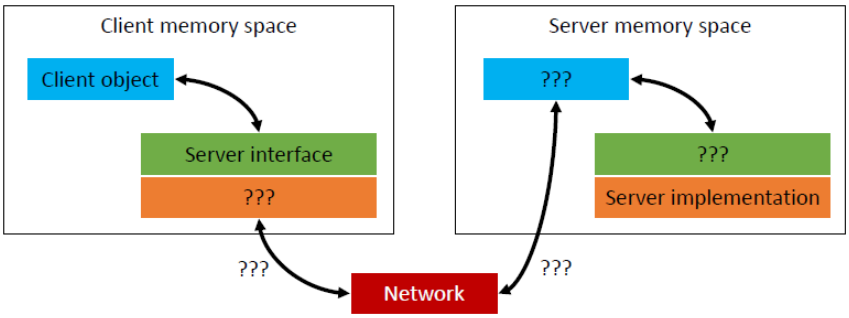
Helyi hívás

- a kliens és a szerver ugyanabban a memóriatérben vannak
- a kliens objektumnak közvetlen mutatója van a szerver objektum memóriacímére
- a hívás tehát közvetlenül megtörténhet
- a kliens a szerver publikus interfészén valamilyen függvényt meghív, ez a függvény a szerver implementációban lefut, és az eredménye közvetlenül visszakerül a klienshez



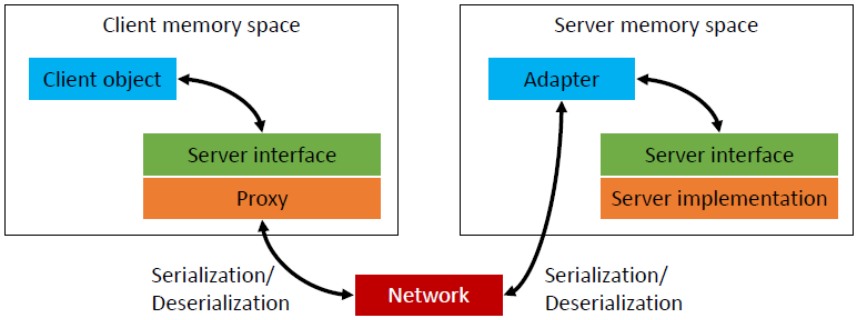
Távoli hívás

- elosztott eset, tehát távoli hívás estén a kliens objektum és a szerver objektum külön memóriatérben vannak
- a kliensnek nincsen közvetlen mutatója a szerver objektumra, így a két objektumnak valamilyen hálózati kapcsolaton keresztül kell kommunikálnia egymással
- a cél az lenne, hogy a kliens számára ne tűnjön fel az, hogy ő egy távoli objektummal kommunikál
- éppen ezért, neki ugyanolyan interfészt kell látnia, mintha helyi interfészt látna



- ezt az interfészt valójában egy Proxy objektum implementálja, amely a kliens hívását hálózati üzenetté konvertálja (ezt a konverziót nevezzük sorosításnak)
- a hálózati üzenetet a szervet oldalon egy adapter fogadja
- az adapter visszasorosítja az üzenetet, majd értelmezve azt, az üzenetet továbbítja a szerver interfészen keresztül a szerver objektumnak

- a szerver függvénye lefut, az eredménye pedig visszakerül az adapterhez
- a visszatérési értéket az adapter sorosítja, és a hálózaton keresztül visszaküldi a Proxy-nak
- a Proxy visszasorosítja az üzenetet, majd létrehozza az eredmény objektumot a memóriában, és visszaadja azt a kliensnek



- a kliens az egészből annyit lát, mintha egy helyi szerverobjektummal kommunikált volna

Távoli hívás felmerülő problémái

- érdemes azonban belegondolni, hogy azért nem ennyire egyszerű a dolog
- a Proxynak például meg kell találnia valahogy a szervert
- az adapternek pedig demultiplexálnia kell a szerver függvényei között, sőt, ha több szerver objektum van, akkor a szerver objektumok között is, és az adapternek egyszerre több kliens kérést is ki kell tudnia szolgálni
- a szerver oldalon tehát a többszálúsággal is foglalkozni kell
- ugyancsak kérdés, hogy hogyan sorosítsuk az üzeneteket, és hogyan oldjuk meg ezt az egész kommunikációt akkor, hogyha a kliens és a szerver különböző programnyelveken van írva
- például mit jelent ilyenkor a szerver interfész
- ezek mind-mind olyan problémák, melyek egy helyi hívásnál nem merülnek fel

Elosztott OO kapcsán felmerülő kérdések

Hogyan definiáljuk a szerver interfészét, ha …

1. ugyanaz a programnyelv?
    - amennyiben a szerver és a kliens ugyanazon a programnyelven készül, akkor használhatjuk a programnyelv interfész fogalmát
    - így működik például a Java RMI, vagyis Remote Method Invocation, ahol a szerver interfészét egy Java interfész adja meg
    - hasonlóan a .NET RMI-ben a szerver interfészt egy .NET-es interfész írja le
2. különböző a programnyelv?
    - ha azonban a kliens és a szerver különböző programnyelvekben, illetve különböző keretrendszerekben fut, akkor már nem ennyire egyszerű a helyzet
    - ilyenkor valamilyen programnyelvektől független interfész leíróra van szükségünk, amely leképezhető a különböző programnyelvekre
    - hamarosan majd tanulunk ilyen megoldásokat

A kliens hogyan találja meg a szervert, ha …

1. a kliens tudja, hol van a szerver?
    - ez tipikusan azt jelenti, hogy a kliensnek valamilyen konfigurációs fájljában szerepel a szervernek a címe
2. a kliens csak egy logikai nevet ismer a szerverből?
    - ezt a logikai nevet valakinek fel kell oldania, és le kell fordítania fizikai címmé
    - ezt a fordítást egy úgy nevezett naming service, vagyis névszerver végzi
    - fontos megjegyezni, hogy a logikai név itt nem a DNS és a fizikai cím nem az IP cím, hanem ennél egy sokkal általánosabb dologról van szó
    - a logikai név tipikusan valamilyen karakterlánc, a fizikai név pedig tipikusan egy URL
    - de ettől eltérő megoldások is elképzelhetőek
3. a kliens csak egy interfészt ismer?
    - olyan szervert keres, amely ezt az interfészt implementálja
    - az úgynevezett trading service az, amely egy interfész specifikációt fizikai címmé képez le
    - például, ha a kliensnek az időjárásra van szüksége, akkor a trading service-nek csak annyit mond, hogy egy időjárás-jelentést szolgáltató interfészre van szüksége
    - a trading service, pedig ezek közül választ egyet, és annak a címét küldi vissza kliensnek
    - a kliens számára mindegy, hogy melyik szerver implementációt kapja, számára csak annyi érdekes, hogy a szerver támogassa az általa elvárt interfészt

Hogyan implementáljuk a proxy-t?

- A proxy feladatai:
    - szerver megkeresése
    - kapcsolat felépítése
    - input paraméterek és a visszaérkező eredmény sorosítása
- a proxy megvalósításához általában a proxy és az adapter tervezési mintát használjuk
- hogyha valamilyen szabványos kommunikációról van szó, akkor a keretrendszerek általában biztosítják számunkra a proxy objektumot, így azt nem kell nekünk kézzel megírni

Hogyan implementáljuk az adaptert?

- Az adapter feladatai:
    - kliens kapcsolatok fogadása
    - szerver objektumok példányosítása
    - a klienstől érkező kérések demultiplexálása a megfelelő szerver objektum megfelelő függvényéhez
    - a beérkező paraméterek és az eredmény sorosítása
    - több kérés kiszolgálása párhuzamosan
- a megvalósítás tipikusan az adapter tervezési mintával történik
- hasonlóan a proxy-hoz, hogyha szabványos kommunikációról van szó, akkor a keretrendszerek biztosítják számunkra az adaptert, így azt nem kell nekünk kézzel megírnunk

Hogyan sorosítsuk az adatokat?

- a sorosítás általában megint csak nem kézzel történik, hanem a keretrendszer automatikusan elvégzi helyettünk
- Java és .NET esetén a megfelelő annotációkat használva az objektumokat XML-é vagy JSON-á lehet sorosítani, és onnan visszasorosítani
- fontosa megjegyezni, hogy a sorosítás mindig deep copy-t jelent, vagyis a teljes objektum hierarchiát ki kell írnunk, a másik oldalon pedig a teljes hierarchiát kell rekonstruálnunk
- éppen ezért a pointerek, referenciák, in/out, illetve az out paraméterek egészen máshogy működnek, mint egy helyi hívás esetén
- a sorosítás a kommunikációs technikától függően működhet bináris, illetve szöveges formátumban

Bináris sorosítás

- a bináris sorosítás általában gyors és hatékony, kevés memóriát használ, azonban a kliens és a szerver között bináris kompatibilitást feltételez
- a bináris sorosítást általában akkor használjuk, ha a kliens és a szerver ugyanazon a programnyelven készül
- így működik például a Java RMI és a .NET RMI

Szöveges sorosítás

- szöveges sorosítás esetében a sorosított objektumok általában emberileg is olvashatók
- például XML vagy JSON formátumúak
- ez a fajta megoldás azonban lassú és nem túl hatékony, továbbá elég sok memóriát igényelhet
- van azonban egy hatalmas előnye a bináris sorosításhoz képest, méghozzá az, hogy nagyon jó kompatibilitást biztosít akár programnyelvek között, akár pedig az idők folyamán
- a bináris sorosításnál ugyanis foglalkozni kell a byte sorrenddel, a processzor architektúrájával, amelyek az idők folyamán változhatnak
- a szöveges sorosításnál ilyen problémák nem lépnek fel

Hogyan kezeljük a memóriát?

- a kérdés tulajdonképpen arról szól, hogy ki foglalja le és ki szabadítja fel a bemenő paramétereket, és a visszatérési értékeket
- ha a programnyelv és a környezet támogatja a Garbage Collectort, akkor ezek nem merülnek fel problémaként
- c++-ban azonban nem ilyen egyszerű a helyzet
- kliens oldalon:
    - a bemenő paramétereket tipikusan a kliens foglalja, és ő is szabadítja fel
    - az eredmény objektumot viszont a proxy szabadítja fel, de a kliensnek kell felszabadítania
- szerver oldalon:
    - az adapter foglalja le a memóriaterületet a bemenő paraméterek számára a visszasorosítás során
    - az eredményt azonban a szerver oldal foglalja, de az adapternek kell felszabadítania

- nagyon pontosan követni kell tehát azt, hogy ki a tulajdonosa egy adott objektumnak, és kinek a felelőssége felszabadítani azt
- jól látható, hogy a memóriakezeléssel kapcsolatos feladatok felszivárognak egy magasabb absztrakciós szintre
- ennek köszönhetően a kliens számára nem teljesen transzparens, hogy egy helyi, vagy egy távoli objektummal kommunikál

Hogyan szolgálunk ki több klienst …

- egy szálú szerver esetén?
    - az egy szálú szervernek az a legnagyobb előnye, hogy nem léphetnek fel konkurencia problémák, így nem is kell bonyolult többszálúsággal és konkurenciával foglalkozni
    - az egyszálú szerver implementációja lehet blokkoló
        - ez azt jelenti, hogy a kéréseknek várakoznia kell, amíg az aktuális kérést a szerver ki nem szolgálta
        - ilyen szervert általában nem szoktunk készíteni, mert ilyenkor a klienseknek általában nagyon sokat kell várakozniuk, és a szervergép erőforrásainak kihasználtsága sem túl hatékony
    - a másik megoldás az egyszálú, de nem blokkoló szerver implementáció
        - így működik például a node.js
        - ebben a modelben a szerver kiszolgáló függvényei rövidek, és amint valamilyen hosszú, blokkoló műveletre van szükség, az a háttérben hajtódik végre, és amint az eredmény rendelkezésre áll, a kérés kiszolgálása egy callback függvényben folytatódik
        - közben a szerver újabb kéréseket szolgálhat ki, vagy más callback függvényeket futtathat le
        - ez a megoldás nagyon hatékony tud lenni erőforrás-kihasználtság szempontjából, több kérést is ki tud egyszerre szolgálni
        - azonban a hátránya, hogy egy komplex, callback függvényeken alapuló programozási modellt igényel
- többszálú szerver esetén?
    - egyik megoldás, hogy minden kliens önálló szálat kap
        - de ez gyorsan a szerver túlterheléséhez vezethet
    - a másik megoldás a Thread pool
        - ilyenkor a szervernek van egy fix készlete a kiszolgáló szálakból, amit egy Thread pool menedzsel, és amíg van szabad szál, a kliens kéréseket ki tudja szolgálni
        - ha minden szál foglalt, a kliens kéréseknek várakozniuk kell
        - ennek a megoldásnak az az előnye, hogy jól ki tudja használni a szerver erőforrásait, vagy ha ez nem történne meg, akkor a Thread pool méretét dinamikusan lehet növelni, vagy csökkenteni az erőforrás-kihasználtság optimalizálásához
        - a többszálú megoldásoknak azonban van egy nagyon-nagy hátránya, méghozzá az, hogy a több szál által használt közös erőforrásokat védeni kell, nehogy valamilyen inkonzisztens állapot lépjen fel egy versenyhelyzet miatt

        - a többszálú programozás meglehetősen nehéz feladat, és könnyű hibákat véteni benne

Hány objektum példány kell a szerverből?

- egy
    - az egyik lehetőség, hogy egyetlen szerver példány van, lényegében egy singleton, és minden kliens kérést ez szolgál ki
    - ez a megoldás akkor működhet, ha nincsen kliensfüggő állapot
- kliensenként egy
    - egy másik megoldás, hogy minden kliens önálló szerverobjektumot kap
    - ez az objektum kliensspecifikus állapotot is tud tárolni, azonban ez nem egy skálázható megoldás
    - ha például a szervergép újraindul, akkor ez az állapot elveszhet, ha pedig a szervergép kiesik, egy másik gép nem tudja átvenni ennek a szerepét
    - további kérdés, hogy mikor lehet megszüntetni a szerver objektumot
    - általában szükség van a kliens objektumtól egy lezáró hívásra, de ha egy ilyen hívás elveszik, az memory leak-hez vezet
- Object pool
    - egy másik lehetőség, hogy van egy Object pool a kiszolgáló szerver objektumokból, és ezekhez rendeljük hozzá a beérkező kéréseket
    - így a kérések egymástól szeparálva futnak le, de kliens specifikus állapot nem tárolható hozzájuk a szerver memóriájában
    - ennek a megoldásnak az előnye az, hogy skálázható, nem baj, ha újraindul a szervergép, illetve a terhelés növekedése újabb szervergépek indításával kezelhető

Hogyan őrizzük meg az állapotot a hívások között?

- szerver memóriájában
    - az egyik lehetőség, hogy a szerver a memóriájában tárolja az állapotot
    - ehhez az kell, hogy minden kliens saját szerverobjektummal rendelkezzen
    - ahogy már megbeszéltük, ez a megoldás nem skálázható
- minden hívásban átküldjük
    - egy lehetséges megoldás az, hogy a kliens mindig elküldi a teljes állapotot a hívással együtt
    - ez egy működő és skálázódó megoldás, de csak akkor, ha ez az állapot nem túlságosan nagy méretű
    - ezzel a módszerrel működnek például a Cookie-k a böngészőkben
- adatbázisban
    - egy lehetséges megoldás az, hogy az állapotot egy adatbázisban tároljuk
    - ez is jól skálázódik, azonban a kliensnek azonosítani kell tudnia magát, hogy a szerver tudja, hogy adott kérés esetén a kliensről milyen információt kell előbányásznia az adatbázisból
    - általában ez azt jelenti, hogy a kommunikációhoz tartozik egy Session ID, amelyet minden hívásba belerak a kliens
    - ezt a Session ID-t a kliensnek kell tárolnia, amely történhet például Cookie-kban a böngészők esetén

Hogyan kommunikáljunk úgy, ha valamelyik oldal nem elérhető?

- szinkron hívások esetén, amikor a kliensnek azonnal kell a válasz, akkor nyilvánvalóan nem tudunk kommunikálni
- aszinkron esetben azonban, ha van egy megbízható köztese szereplő, aki ideiglenesen tudja tárolni a kéréseket, akkor megoldható a szétcsatolt kommunikáció
- az aszinkron megoldás nehézsége azonban az, hogy a visszaérkező válaszokat valahogyan párosítani kell a korábban kiküldött kérésekkel

Hogyan kezeljük a szinkron hívásokat?

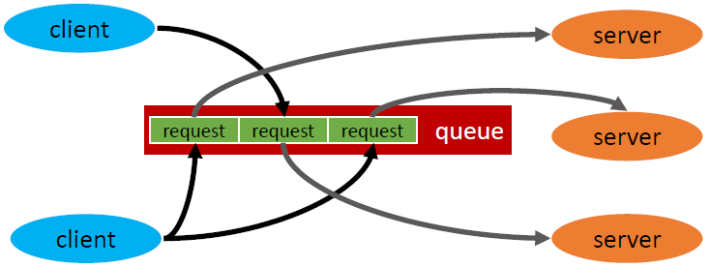
- ahogy már említettük, itt egy folyamatosan fennálló kapcsolatra van szükség
- a kliens elküldi ezen keresztül a kérését, majd blokkolva vár a szerver válaszára
- a szerver kiszolgálhatja a kérést szinkron és aszinkron módon is
- szinkron módon működnek például a hagyományos Java alapú szerverek
- aszinkron módon szolgál ki például a Node.js

Hogyan kezeljük az aszinkron hívásokat?

- amennyiben nincsen megbízható köztes szereplő,
    - a kiszolgálás történhet úgy, mint az előző szinkron esetben
    - a különbség csak kliens oldalon van: a kliens nem várakozik blokkolva, hanem vagy egy callback híváson keresztül értesül az eredményről, vagy periodikusan poll-ozza a szervert, hogy elkészült-e már az eredmény
- amennyiben van megbízható köztes szereplő,
    - a kliens és a szerver nem ismeri egymást közvetlenül, hanem szétcsatolva, valamilyen üzenetsoron keresztül kommunikálnak
    - ebből a megoldásból általában kétfajta modell van
    - az egyik üzeneteken alapul, ahol sok kliensünk van, amelyeket egy szerver szolgál ki (kliens-szerver)
    - a másik eseményeken alapul, ahol általában egy forrásunk van, amely eseményeket generál, és sok feliratkozó, akik reagálnak ezekre az eseményekre (publish-subscribe)

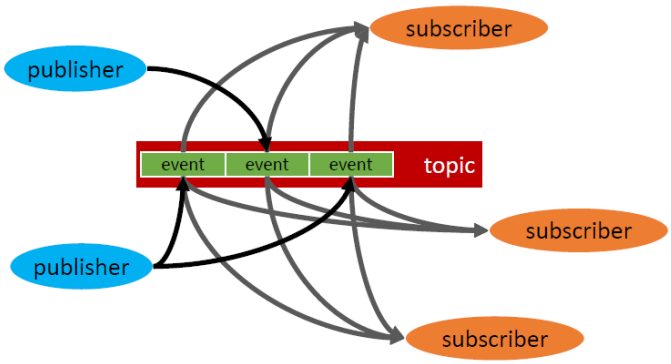
Aszinkron üzenetek: kliens-szerver

- a kliensek üzeneteket küldenek a köztes szereplő által fenntartott üzenetsorba
- a szerver pedig ebből a sorból veszi ki egyesével az üzeneteket
- és a kiszolgálás után esetleg egy másik sorba küldi vissza a választ
- a szerverből több példány is futhat, de ezek a példányok általában egyformák, ugyanazt a feladatot végzik, mindössze csak a skálázhatóság miatt van belőlük több példány
- egy üzenetet csak egy szerverpéldány dolgozhat fel, nem lehet olyan, hogy egy üzenetet több példány is megkap



Aszinkron események: publish-subscribe

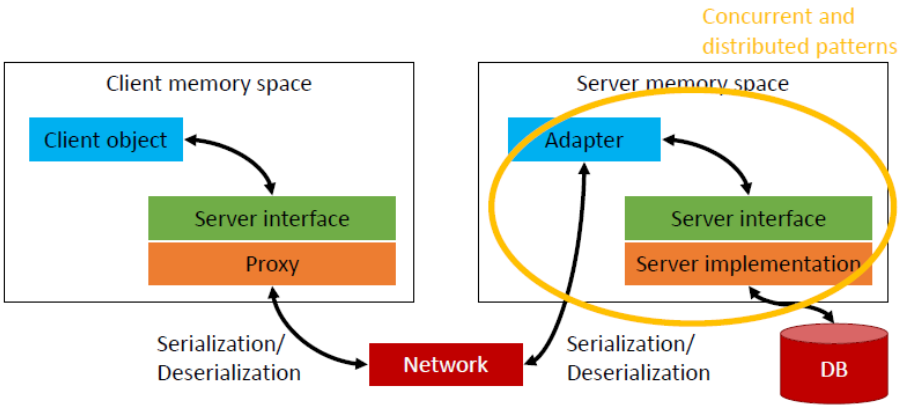
- ez a modell eseményeken alapul
- tipikusan egy termelőnk van, ami eseményeket generál
- ezeket a köztes, megbízható fél egy topic-ban gyűjti, és a fogadók feliratkozhatnak a topic-ban érkező eseményekre
- az előző modellhez képest az a különbség, hogy itt minden eseményt, minden feliratkozó megkap
- általában az is különbség, hogy itt a fogadók nem küldenek válaszüzenetet a beérkező eseményekre



Technológiák elosztott kommunikáció megvalósításához

Konkurens és elosztott minták

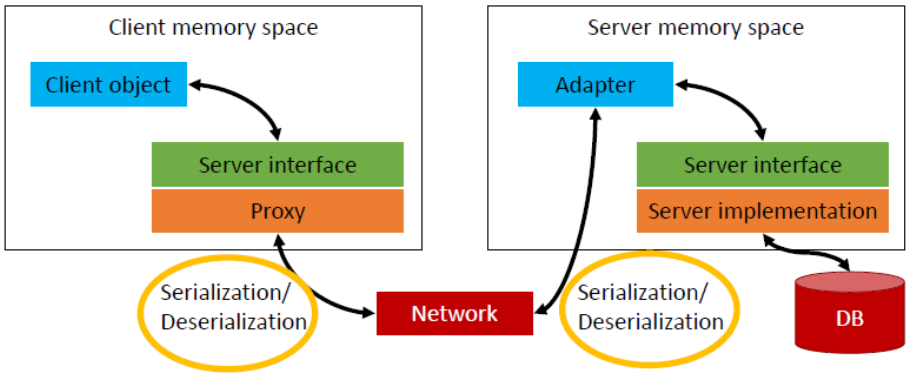
- ez a többszálú szerver megvalósításához kapcsolódik
- biztosítani kell a szálak közötti szinkronizációt, a szálak által közösen használt erőforrásokra kölcsönös kizárást, és le kell tudni kezelni a kliensektől beérkező kéréseket
- ezekkel a kérdésekkel foglalkoznak a konkurens és elosztott minták
- ezt a témát majd egy későbbi előadásban fogjuk körüljárni



Sorosítás

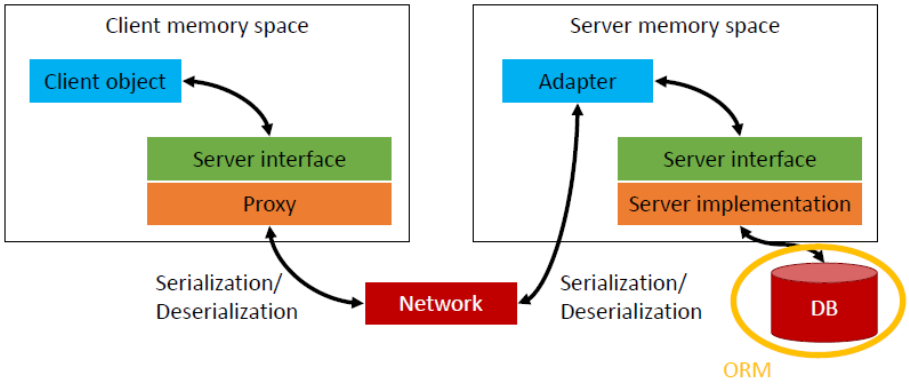
- bináris sorosítást .NET-ben és Javaban a legegyszerűbeen a hagyományos objektum sorosítással lehet megoldani
- ehhez Javaban a Serializable interfészt kell implementálni, .NET-ben pedig a Serializable attribútumot kell használni

- szöveges sorosításra (vagyik objektumok XML vagy JSON fájlra való konvertálására) Javaban a JAXB annotációk, .NET-ben a WCF-es DataContract annotációk használhatók



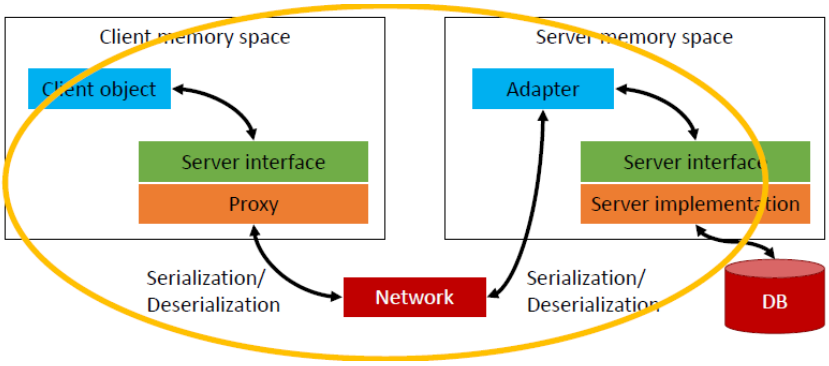
Objektum-relációs leképezés

- a szerveroldali adat tárlására adatbázisok használhatók
- objektumorientált esetben ezekhez valamilyen objektum-relációs leképezéssel férhetünk hozzá
- Java esetén ennek a megvalósítása tipikusan a JPA, vagyis a Java Persistence API annotációinak használata
- .NET-ben pedig az Entity Framework attribútumainak használata



Kommunikációs technológia és keretrendszer

- az egész kommunikációs technológia megvalósítására is több lehetőség kínálkozik
- ha a kliens és a szerver ugyanolyan keretrendszerrel készül, akkor Java esetén a Java RMI, .NET esetén a .NET RMI használható
- amennyiben a kliens és szerver keretrendszere eltérő, akkor célszerű valamilyen programnyelvektől független, lehetőleg szabványos technológiát használni
- ilyenek lehetnek például az XML-re épülő SOAP webszolgáltatások
- vagy a tipikusan JSON-ra vagy XML-re épülő HTTP alapú REST szolgáltatások

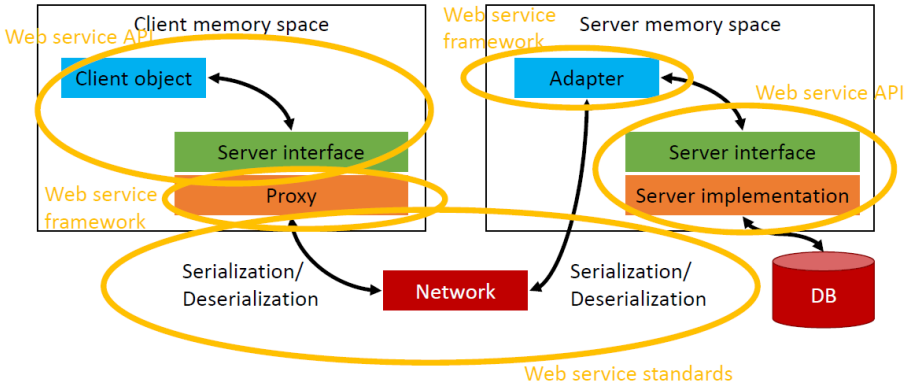


SOAP webszolgáltatások

- a SOAP egy XML-re épülő programnyelvektől és operációs rendszerektől független kommunikációs szabvány
- ma már széles körben elterjedt, és jól támogatott megoldás
- mind .NET-ben, mind Java-ban egy nagyon kényelmes típusos API tartozik hozzá

Kommunikáció SOAP webszolgáltatásokkal

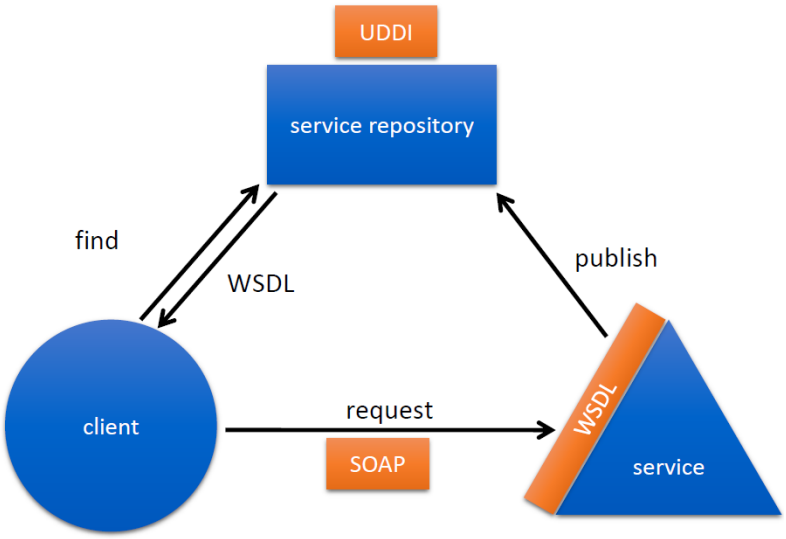
- Az elosztott kommunikációs ábránkra vetítve a SOAP szabvány a legalsó, hálózati réteget definiálja



- a szabvány mindössze annyit mond meg, hogy milyen formátumú XML fájlokat kell egymással cserélgetni
- az XML előállítása és feldolgozása az adott programnyelv és keretrendszer fejlesztőire hárul
- ennek köszönhetően bármely olyan alkalmazás, amely a konkrét XML-t elő tudja állítani, illetve fel tudja dolgozni, része lehet egy ilyen jellegű kommunikációnak
- a SOAP webszolgáltatások esetén a szabványos interfész leíró az XML formátumú WSDL, amely leképezhető a hagyományos programnyelvi interfészekre
- a .NET és a Java is olyan keretrendszerek, amelyek ezt a WSDL-t tudják értelmezni, és .NET illetve Java interfészre tudják azt leképezni
- ugyancsak mindkét keretrendszer képes a Proxy és az Adapter automatikus előállítására is
- nekünk fejlesztőknek, csak a kliens objektumot kell megírnunk, szerver oldalon pedig a WSDL-ből generált szerver interfészt implementálni, az összes többi doboz megvalósításáért a keretrendszer felelős
- a technológia olyan szinten kiforrott, hogy .NET és Java között típusosan lehet hívást intézni, mintha csak egy, a saját programnyelvünkben implementált objektumot hívogatnánk

SOAP webszolgáltatásokhoz kapcsolódó technológiák

- A SOAP webszolgáltatásokhoz kapcsolódó technológiákat az alábbi ábra mutatja be:



WSDL = Web-Services Description Language
SOAP = Simple Object Access Protocol
UDDI = Universal Description, Discovery and Integration

- a szolgáltatás a WSDL interfész leíróban adja meg azt, hogy rajta milyen függvények, milyen paraméterekkel hívhatók, és ezen függvények milyen kivételeket dobhatnak
- ezt az intefészleíró WSDL-t lehet publikálni egy szolgáltatáskatalógusban, amely kereshető, és ha a kliens megtalálta a számára megfelelő interfész leírót, akkor annak segítségével egy SOAP üzenettel meghívhatja a szolgáltatást

Példa: SOAP kérés

- tegyük fel, hogy van egy IHelloWorld interfészünk egy SayHello függvénnyel, amely egy string típusú name paramétert vár, és string eredménnyel tér vissza

```
interface IHelloWorld
{
    string SayHello(string name);
}
```

- a szót egy XML alapú kommunikáció és a fenti SayHello függvénynek megfelelő üzenet az alábbi módon néz ki:

```
<s:Envelope xmlns:s=
        "http://schemas.xmlsoap.org/soap/envelope/">
    <s:Body>
        <SayHello xmlns= "http://www.iit.bme.hu/soi">
            <name>me</name>
        </SayHello>
    </s:Body>
</s:Envelope>
```

- az XML gyökerében egy Envelope tag található, alatta egy Body tag, azon belül pedig a SayHello függvénynév mint XML tag, benne pedig annyi gyerek tag, ahány paramétere a függvénynek van
- ha a parméterek összetett típusúak, akkor az XML-en belül még ezeknek is lehet belső struktúrája
- jól látható, hogy viszonylag intuitív egy ilyen XML-nek a szerkezete
- ha terveznünk kellene valamilyen XML alapú kommunikációt, akkor nagy valószínűséggel mi is valamilyen hasonló szerkezethez jutnánk

Példa: SOAP válasz

```
interface IHelloWorld
{
string SayHello(string name);
}
```

- a válasz üzenet a kéréshez hasolnó felépítésű üzenet

```
<s:Envelope xmlns:s=
      "http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
      <SayHelloResponse xmlns=
          "http://www.iit.bme.hu/soi">
        <SayHelloResult>Hi: me</SayHelloResult>
      </SayHelloResponse>
  </s:Body>
</s:Envelope>
```

A WSDL összetétele

- SOAP webszolgáltatásoknál a WSDL adja meg az interfészt
- az alábbi kódrészletekben az IHelloWorld interfésznek megfelelő WSDL-t fogjuk látni

```
interface IHelloWorld
{
string SayHello(string name);
}
```

- a WSDL gyökere egy definitions nevű xml tag
- ebben általában a felhasznált xml névterek perfixeit szoktuk definiálni

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions
  targetNamespace="http://www.iit.bme.hu/soi"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:tns="http://www.iit.bme.hu/soi"
  xmlns:ns0="http://www.iit.bme.hu/soi">
  ...
</wsdl:definitions>
```

- a definitions alatt az leső XML tag a types
- ebben a szekcióban XML séma segítségével adjuk meg a saját összetett típusainkat, de minden függvényhez egy bemeneti és kimeneti üzenetstruktúrát is definiálunk
- itt éldául a SayHello függvényhez tartozó beérkező üzenet struktúrája látható:

```
...
 <wsdl:types>
    <xs:schema targetNamespace="http://www.iit.bme.hu/soi"
        xmlns:tns="http://www.iit.bme.hu/soi"
        xmlns:ns0="http://www.iit.bme.hu/soi"
        elementFormDefault="qualified">
      <xs:element name="SayHello" nillable="true"
                  type="ns0:SayHello"/>
      <xs:complexType name="SayHello">
        <xs:sequence>
          <xs:element name="name" type="xs:string"
                      nillable="true"/>
        </xs:sequence>
      </xs:complexType>
```

- itt pedig a válasz üzenet formátumának leítója:

```
...
      <xs:element name="SayHelloResponse" nillable="true"
                  type="ns0:SayHelloResponse"/>
      <xs:complexType name="SayHelloResponse">
        <xs:sequence>
          <xs:element name="SayHelloResult" type="xs:string"
                      nillable="true"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </wsdl:types>
...
```

- a types nevű szekció után a WSDL-ben a message-ek következnek
- ezek adják meg azt, hogy a szolgáltatás milyen üzeneteken keresztül tud kommunikálni
- az üzenet formátumát a types szekcióban definiált valamely típus írja le
- egy ilyen típusra hivatkozik a part tag-en belül az element attribútum

```
...
  <wsdl:message name="IHelloWorld_SayHello_InputMessage">
    <wsdl:part name="parameters" element="ns0:SayHello"/>
  </wsdl:message>

  <wsdl:message name="IHelloWorld_SayHello_OutputMessage">
    <wsdl:part name="parameters„
               element="ns0:SayHelloResponse"/>
  </wsdl:message>
 ...
```

- a WSDL következő része a portType
- leginkább egy portType felel meg egy hagyományos programnyelvben az interfész fogalmának
- a portType-on belül definiáljuk az operációkat, vagyis a szolgáltatás függvényeit
- mindegyiknél megadjuk, hogy mi a bemenő üzenet, mi a kimenő üzenet, és opcionálisan azt is, hogy milyen lehetséges kivétel üzenetek keletkezhetnek

```
...
  <wsdl:portType name="IHelloWorld">
    <wsdl:operation name="SayHello">
      <wsdl:input wsaw:action=
         "http://www.iit.bme.hu/soi/IHelloWorld/SayHello"
         message="ns0:IHelloWorld_SayHello_InputMessage"/>
      <wsdl:output wsaw:action=
   "http://www.iit.bme.hu/soi/IHelloWorld/SayHelloResponse"
         message="ns0:IHelloWorld_SayHello_OutputMessage"/>
    </wsdl:operation>
  </wsdl:portType>
 ...
```

- a WSDL következő része a binding szekció, amely a szolgáltatás protokolját konfigurálja
- erre azért van szükség, mert a SOAP protokollnak többféle verziója is létezik
- sőt, lehetőség van digitális aláírás és titkosítás konfigurálására is
- de a WSDL elég általános ahhoz, hogy a SOAP-on kívül akár más protokoll-okat is fel tudjunk vele konfigurálni

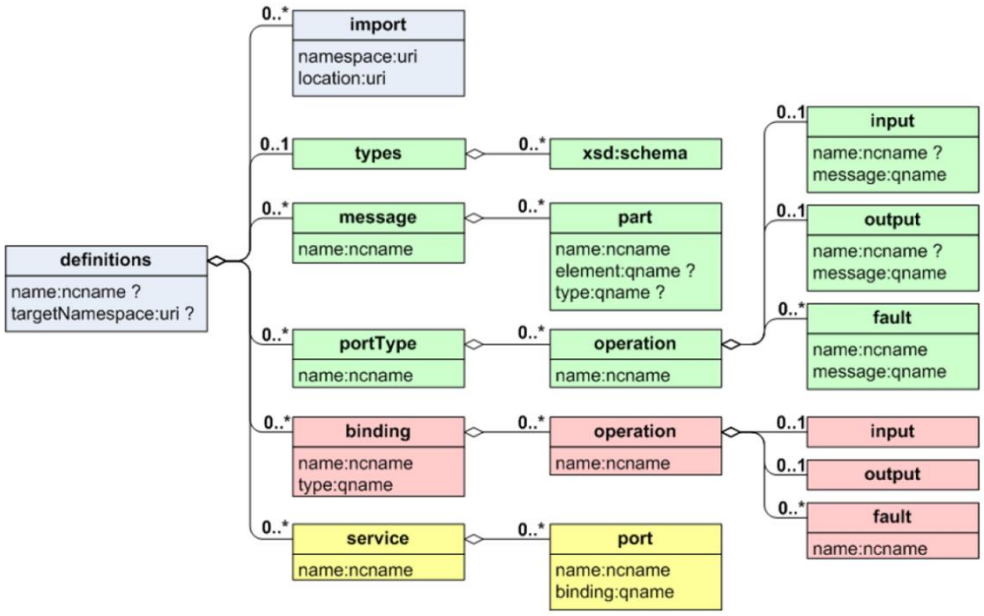
```
...
   <wsdl:binding name="IHelloWorld_BasicHttpBinding_Binding"
                 type="ns0:IHelloWorld">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="SayHello">
      <soap:operation style="document" soapAction=
       "http://www.iit.bme.hu/soi/IHelloWorld/SayHello"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
...
```

- a WSDL legutolsó szekciója a service, amely belül megadhatjuk, hogy a szolgáltatás milyen URL-en keresztül érhető el
- erre az URL-re HTTP kapcsolaton keresztül kell elküldeni a SOAP üzenetet, és ezen a kapcsolaton keresztül kapjuk vissza a SOAP választ

```
...
  <wsdl:service name="HelloWorld">
    <wsdl:port name="IHelloWorld_BasicHttpBinding_Port"
      binding="ns0:IHelloWorld_BasicHttpBinding_Binding">
      <soap:address location=
          "http://www.iit.bme.hu/Services/HelloWorld"/>
    </wsdl:port>
  </wsdl:service>
```

WSDL összegzés

- a WSDL összetételét az alábbi ábra foglalja össze:

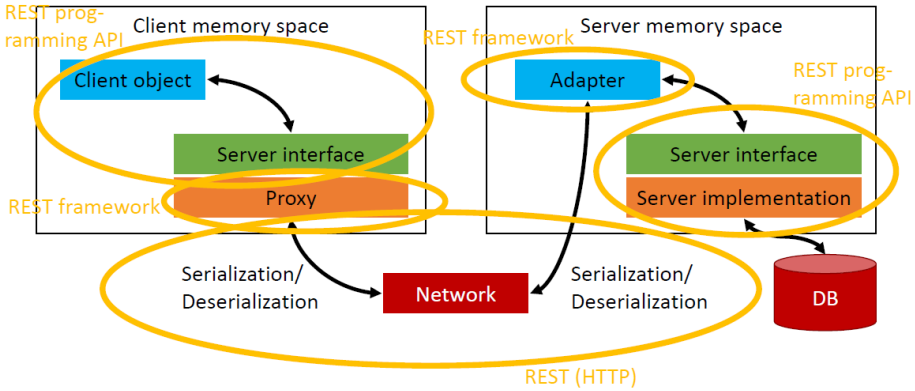


- a types szekcióban adjuk meg a saját összetett típusainkat
- a message-ekben definiáljuk a szolgáltatás által elfogadott és visszaküldött üzeneteket
- a portType definiálja a szolgáltatás interfészét, vagyis a rajta meghívható függvényeket
- a binding konfigurálja a protokollt
- a service pedig tartalmazza a konkrét URL-t, ahol a szolgáltatás meghívható
- ezen kívül használhatjuk az import tag-et, melynek célja az, hogy a WSDL-t kisebb részekre tudjuk feldarabolni
- ezáltal a WSDL részei modularizálhatók

REST szolgáltatások

REST szolgáltatások

- a REST a HTTP protokoll kibővítése (RESTful HTTP)
- REST = REpresentetional State Transfer
- az elosztott kommunikációs ábránkon a REST a hálózatos kommunikációkért felelős

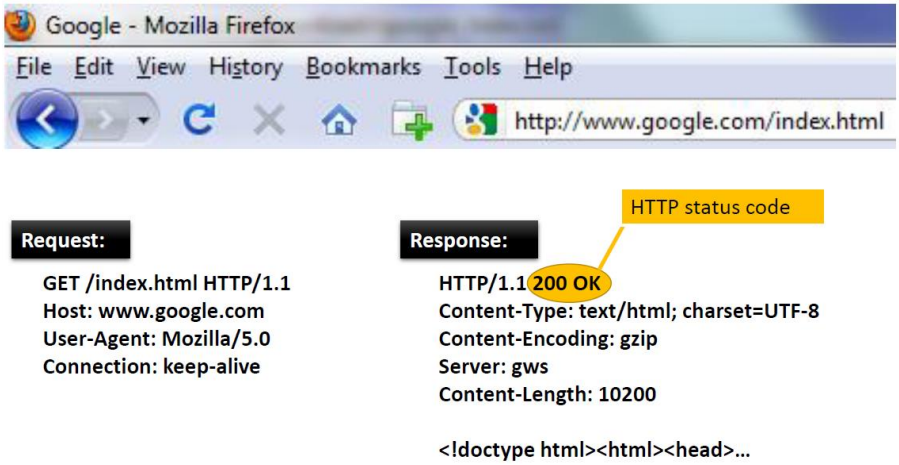


- itt is, akár csak a SOAP esetén, a magasabb rétegek megvalósításáért a keretrendszerek felelnek
- bármely alkalmazás tud REST-en keresztül kommunikálni, hogyha képes HTTP protokoll kezelésére
- .NET és Java esetén a Proxy-t és az Adaptert a keretrendszer biztosítja

- a REST-hez a technológia kezdetén nem volt általános interfész leíró, de manapság a Swagger vagy szabványosított nevén OpenAPI használható REST szolgáltatások interfészének leírására
- ebből az interfész leírásból generálhatók Java, illetve .NET-es interfészek
- nekünk fejlesztőknek továbbra is elegendő a kliens objektumot és a szerver implementációt megírni
- minden mást a keretrendszer elintéz

HTTP GET kérés és válasz

- a REST a HTTP protokoll kibővítése
- vizsgáljuk meg röviden, hogyan is működik a http

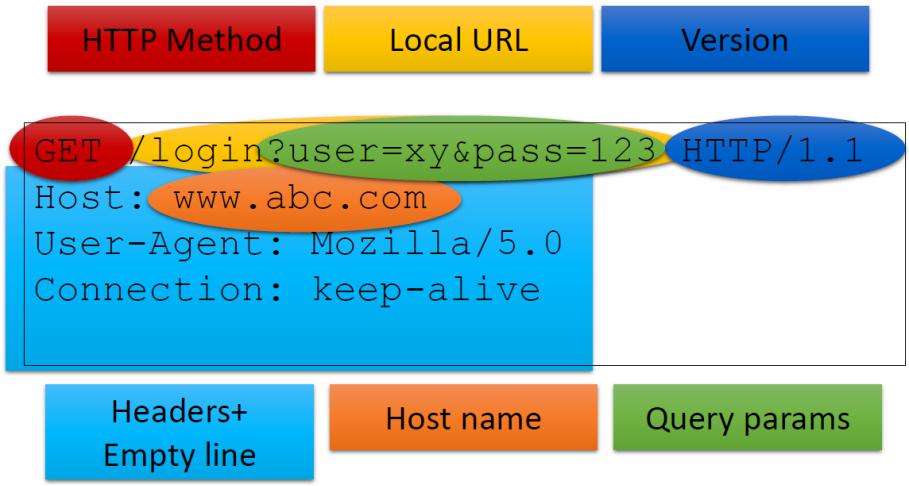


- amikor egy böngészőben beírjuk egy böngésző címét, akkor a böngésző a domain név alapján feloldja a szerver IP címét
- ennek az IP címnek a 80-as portjára nyit egy TCP kapcsolatot
- majd ezen keresztül szövegesen elküldi a HTTP kérést
    - a HTTP kérés első sorában szerepel egy ige
    - tipikusan a GET, vagyis, hogy szeretnénk lekérni egy weboldal tartalmát
    - ezt követi a böngészőbe beírt URL lokális része, amely a szerveren belül azonosítja a weboldalt
    - az első sort a használni kívánt HTTP protokoll verziója zárja
    - a következő néhány sorban a HTTP fejrészek következnek
    - ezek név-érték párok, melyeket kettőspont választ el egymástól
    - a fejlécek felsorolásának végét egy üres sor zárja
    - ezután következhet a HTTP kérés törzse, amely GET művelet esetén tipikusan üres
- a szerver szintén egy HTTP üzenetet küld vissza
    - válaszként a szerver visszaküldi a választott HTTP verziót, egy státuszkódot, és a státuszkód szöveges megfogalmazását
    - ezután a sor után szintén HTTP fejlécek következnek, amelyeket ugyancsak egy üres sor zár
    - ezt követően pedig a HTTP válasz törzse következik
    - ebben törzsben szerepel a weboldal tartalma, a fejlécek között pedig a Content-Type adja meg a formátumát
    - ha ez a formátum text/html, akkor a böngésző megjeleníti a weboldalt
    - ha ez a formátum mondjuk egy zip fájlra utal, akkor a böngésző letölti a fájlt

- mindig ettől a Content-Type-tól függ, hogy mit kell kezdeni a szervertől visszakapott tartalommal

HTTP GET kérés kielemzése

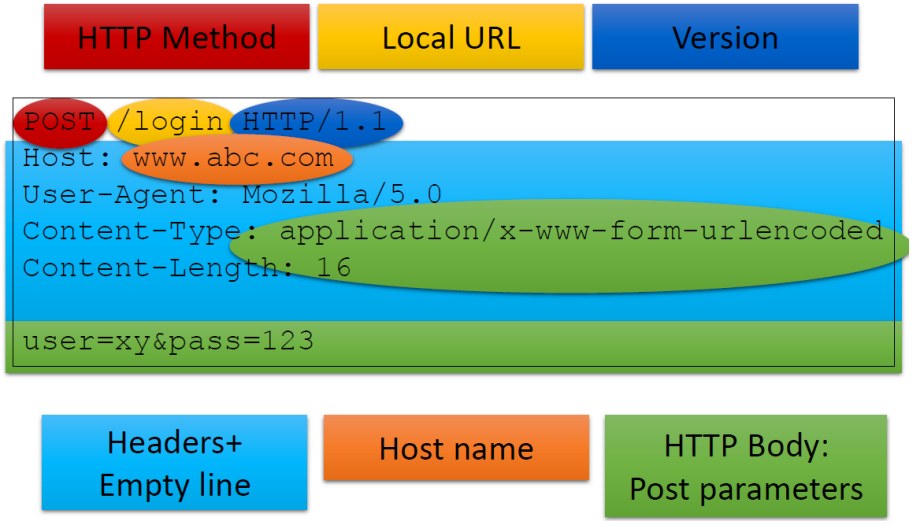
- Példa GET kérés:



- a kérés első sorában először a HTTP metódus szerepel, jelen esetben ez egy GET
- majd ettől szóközzel elválasztva következik az uml lokális része (/login)
- ennek végén kérdőjel után következhetnek a query paraméterek, amelyeket egy & jel választ el egymástól (?user=xy&pass=123)
- a query paraméterek pedig olyan név-érték párok, ahol az értéket a névtől egy egyenlőségjel választja el
- a query paraméterek segítségével dinamikusan paraméterezhető a kérés
- az első sor végén található a HTTP verziója (HTTP/1.1)
- az első sor után következnek a fejlécek név: érték formában
- a fejlécek listáját pedig egy üres sor zárja
- fontos hangsúlyozni, hogy ez csak egy példa, felhasználónév és jelszó soha ne szerepeljen query paraméterben, mert ezek értékét minden köztes szereplő látja

HTTP POST kérés kielemzése

- Példa POST kérés:



- a biztonságos bejelentkezéshez HTTPS kapcsolat kell, és a felhasználónevet és jelszót, csak a HTTP kérés törzsében szabad elküldeni
- erre ad megoldást a POST kérés, amely nagyon hasonlít a GET-hez, a különbség csupán annyi, hogy az igénk az nem GET, hanem POST, a paraméterek pedig nem query paraméterben vannak, hanem a HTTP törzsben, úgynevezett POST paraméterként vándorolnak át
- a HTTP kérés törzse HTTPS kapcsolat esetén titkosított, felhasználó nevet és jelszót csak itt szabad átküldeni, hogy azokat a kliensen és a szerveren kívül más ne láthassa
- a POST kérés még egy lényeges szempontban különbözik a GET kéréstől: a GET mellékhatás-mentes, a POST viszont mellékhatásos
    - például a bankkártyás fizetést POST kéréssel kezdeményezzük
    - biztosan mindenkivel előfordult már az, hogy egy POST kérés után nem töltődött be a weboldal, és abban ráfrissített a böngészőben, akkor böngésző szól, hogy ez egy veszélyes művelet, és megkérdezte, hogy biztosan szeretnénk-e újra küldeni a kérést
    - nyilván egy bejelentkezést minden további nélkül újra lehet küldeni, de egy hitelkártya-tranzakciót már nem érdemes, mert előfordulhat, hogy az újraküldés hatására duplán fogunk fizetni

REST által használt igék

- a GET a keresésnek, illetve a lekérdezésnek felel meg
- a POST egy erőforrás létrehozásának
- a PUT egy erőforrás frissítésének
- a DELETE pedig egy erőforrás törlésének felel meg

A kérések során szervernek több helyen is küldhetünk bemenő paramétert

- paraméterek lehetnek az URL / jel közötti részei
- küldhető paraméter a query string-ben
- küldhető paraméter POST paraméterként
- de küldhető bármilyen információ bármilyen formátumban a HTTP törzsben is
- amikor a szerver válaszol, az eredmény tipikusan a HTTP törzsben kapjuk meg

- a REST nagyon nagy előnye, hogy böngészőből is egyszerűen tesztelhető, nem kell hozzá külön programozott klienst írni
- a következőkben arra nézünk példákat, hogy hogyan is néz ki egy REST API, vagyis tipikusan milyen igékhez milyen URL-ek társulnak, és hogyan adhatóak át a paraméterek

GET példák

- egy REST API esetén fontos megkülönböztetni azt, hogy egy kérés egy gyűjteményre irányul, vagy egy konkrét erőforrásra
- ha az URL végén egy azonosító szerepel, akkor konkrét erőforrásról van szó, ha pedig egy többesszámú név, akkor erőforrások gyűjteményéről
- ha a GET kérést egy gyűjteményre adjuk ki, akkor visszakapjuk a gyűjtemény összes elemét, ha konkrét erőforrásra adjuk ki a kérést, akkor az adott azonosítójú erőforrást kapjuk vissza

- query paraméterekkel lehet jelezni a szerver felé, ha valamilyen szűrést vagy rendezést szeretnénk, hogy a szerver visszaadjon, mielőtt a szerver visszaadja az eredményeket
- Néhány példa:
    - GET /api/movies
        - visszaadja az összes filmet
    - GET /api/movies/12
        - visszaadja a 12-es azonosítójú filmet
    - GET /api/movies/12/actors
        - visszaadja a 12-es azonosítójú filmben játszó színészeket
    - GET /api/movies?orderby=title
        - visszaadja az összes filmet a címük alapján rendezve
- fontos megjegyezni, hogy az alábbi URL-ek csak példák, nem szabványok, mindig csak az adott szerveren múlik, hogy tudja-e ezeket értelmezni
- a szervernek kell definiálni azt az API-t, amit ő támogat, a kliensek pedig majd ehhez fognak alkalmazkodni

POST példa

- egy POST kérést egy gyűjteményre szoktunk kiadni
- a HTTP törzsben pedig elküldjük azt az erőforrást, amit szeretnénk létrehozni
- a szerver létrehozza az erőforrást, hozzárendel egy azonosítót, amelyet válaszban elküldhet a kliensnek is
- a hagyományos POST kéréshez hasonlóan egy ilyen POST kérést is veszélyes megismételni, mert előfordulhat, hogy az erőforrás duplán jön létre, két különböző azonosítóval
- példa új film létrehozására:

```
POST /api/movies

{
"title": "Batman Begins",
"year": 2005,
"director": "Cristopher Nolan"
}
```

PUT példa

- a PUT kéréseket egy konkrét erőforrásra szoktuk kiadni
- a PUT kérés hatására a HTTP törzsben átküldött adatokkal megfrissül az erőforrás
- a PUT kérést szoktuk úgy is értelmezni, hogy ha az adott azonosítójú erőforrás még nem létezik, akkor a szerver ezzel az azonosítóval létrehoz egyet
- ezzel azonban érdemes óvatosan bánni, nehogy két kliens felülírja egymás erőforrásait
- példa a 12-es azonosítójú film frissítésére (illetve, ha még nem létezik, akkor létrehozására):

```
PUT /api/movies/12

{
"title": "Batman Begins",
"year": 2005,
"director": "Cristopher Nolan"
}
```

DELETE példa

- a DELETE műveletet is tipikusan egy konkrét erőforrásra szoktuk kiadni
- hatására törlődik az adott azonosítójú erőforrás
- gyűjteményeken általában nem működik a DELETE, mert tipikusan nem szeretnénk olyan műveletet támogatni, amely az összes erőforrást törli
- Néhány példa:
    - DELETE /api/movies/12
        - törli a 12-es azonosítójú filmet
    - DELETE /api/movies/12/actors/65
        - törli a 65-ös azonosítójú színészt a 12-es azonosítójú filmből

Input paraméter példák

- ahogy mér korábban említettük, input paraméterek sok helyen adhatók be a szerverbe
- érdemes még megemlíteni, hogy a kliens küldhet még egy Accept paraméterű fejlécet, amelyben előírhatja a szerver számára, hogy ő a választ XML-ben, vagy JSON-ben, vagy valamilyen más formátumban szeretné visszakapni
- amennyiben a szerver támogatja a kért formátumot, akkor abban fog válaszolni
- Néhány példa:
    - Query paraméter:
        - http://.../calculator/add?left=3.0&right=5.0
    - Path paraméter:
        - http://.../calculator/add/3.0/5.0
    - Mátrix paraméter:
        - http://.../calculator/add;left=3.0;right=5.0
    - POST paraméter:
        - http://.../calculator/add
        - mint a query paraméterek, csak a HTTP törzsében
    - HTTP törzs:
        - sorosított erőforrás (pl.: XML vagy JSON)
    - Accept fejléc
        - XML: application/xml
        - JSON: application/json

Eredmény példák

- válaszként a szerver az eredményt a HTTP törzsben küldi vissza, amely adott esetben üres is lehet
- az eredmény formátumát pedig a Content-Type fejléc határozza meg
- amennyiben a kliens a kérésben az Accept fejlécben kért valamit, akkor a Content-Type ennek fog megfelelni
- de ha a szerver ezt a formátumot nem támogatja, akkor a szerver hibával, egy 406-os státuszkóddal fog visszatérni

Interfészleíró példa

- REST szolgáltatások estén az interfészleíró az OpenAPI lehet, amely JSON vagy YAML formátumban tartalmazza a szerveren meghívható REST API specifikációját

```yaml
swagger: "2.0"
info:
  title: Sample API
  description: API description in Markdown.
  version: 1.0.0
host: api.example.com
basePath: /v1
schemes:
  - https
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in Markdown.
      produces:
        - application/json
      responses:
        200:
          description: OK
```

Konkurens és elosztott minták

A konkurencia és az elosztott rendszerek problémái

- A többszálú programozás és az elosztott kommunikáció nem könnyű feladatok
- a konkurens és elosztott minták abban segítenek, hogy ezeket korrekt módon tudjuk megoldani

A konkurens programozás előfordulása

- A konkurens programozás legfőbb problémái abból erednek, hogy több szál próbál egy módosítható közös állapothoz egyszerre hozzáférni
- hogyha az állapot csak olvasható lenne, akkor konkurencia problémák nem lépnének fel, hiszen mindenki ugyanazt az értéket látná, és senki nem tudná azt módosítani
- ha nem lenne közösen megosztott állapot, hanem minden szálnak külön állapota lenne, amelyre más szálak nem látnak rá, akkor sem merülnének fel konkurencia problémák, hiszen a szálak nem kommunikálnának egymással
- konkurenciaproblémához tehát egyedül az vezet, hogyha módosítható közös állapot van, amelyhez több szál egyszerre próbál hozzáférni

A konkurens programozás problémái

- konkurens programozás esetben versenyhelyzet van a szálak között, felülírhatják egymás értékeit, így valamilyen módon szinkronizálni kell közöttük, például kölcsönös kizárással
    - de vigyázni kell, nehogy a szálak dead-lock-ba kerüljenek, és el kell kerülni azt is, hogy egyes szálakat kiéheztessünk
- további gondot okoz a párhuzamos programozásnál, hogy az időzítéseket nem lehet pontosan reprodukálni, és nagyon nehéz pontosan kitesztelni a rendszer működését
- további problémákat okoz az, hogy a konkurencia megfelelő kezelése rendkívül komplex feladat, és könnyű hibát véteni benne
    - éppen ezért fontos, hogy tisztában legyünk azokkal a mintákkal, amelyek segíthetnek megoldani ezeket a konkurencia problémákat

Elosztott rendszerek problémái

- az elosztott kommunikáció is sokkal komplexebb, mint a lokális hívások
- problémát okoz a heterogenitás, vagyis, hogy a szerver és kliens különböző programnyelveken is készülhet
- meg kell oldani a transzparenciát, hogy lehetőleg a kliens ne tudjon róla, hogy éppen egy lokális, vagy egy távoli objektummal kommunikál
- a kérések feldolgozása történhet szinkron és aszinkron módon is
- memória menedzsment problémák is felmerülhetnek, hogy kinek kell lefoglalnia, és kinek kell felszabadítania egy adott objektumot
- a hatékony működéshez elengedhetetlen a cache-elés, azonban ilyenkor figyelni kell a másolatok konzisztenciájára
- figyelembe kell venni, hogy a távoli hívásoknak van egy késleltetése
- és a skálázhatóság szempontjából az sem mindegy, hogy a rendszer hol tárol állapotot

- ugyancsak felmerülhetnek versenyhelyzetek, ha több kliens egyszerre használja a rendszert
- figyelni kell a biztonságra, nehogy valaki más nevében férjen hozzá adatokhoz
- fel kell készülni, és reagálni kell tudni a hálózati hibákra, például az üzenetek újraküldésével
- és további gondot okoz, hogy egy elosztott rendszert nagyon nehéz monitorozni, és látni, hogy éppen a rendszer melyik részében mi történik
- egy elosztott rendszer debuggolni is nagyon nehéz, mert a távoli objektumok belső állapotába nem látunk bele

Szinkronizációs minták

Szinkronizációs minták fajtái

- a szálak közötti szinkronizációra nyújtanak megoldást
- négy nagy csoportra oszthatók
    - az első csoport a kritikus szakaszokkal foglalkozik, vagyis olyan műveletekkel, melyek több lépésből állnak, de mégis kívülről atominak tűnnek
    - a második csoport a balking, vagyis a várakozás jellegű minták, amelyek során arra várunk, hogy egy objektum a megfelelő állapotba kerüljön, mielőtt egy műveletet végrehajtunk rajta
    - a harmadik csoportba a jelzések tartoznak, amelyek arról szólnak, hogy hogyan tudnak a szálak egymás között értesítéseket küldeni
    - a negyedik csoport az objektumok publikus interfészével foglalkozik, hogyan lehet megőrizni a hatékonyságot, és rekurzió során elkerülni a dead-lock-ot

Szinkronizációs minták első csoportja: kritikus szakasz

Atomic operations

- az atomi operáció processzor szinten is atomi művelet
- ilyen művelet lehet egy számláló megnövelése, egy referenciának való értékadás, de akár egy feltételes értékadás is
- ilyen műveleteket támogat a C# és a Java is
- az ilyen műveletek azért fontosak, mert drága lock-oláson alapuló kölcsönös kizárás nélkül is működnek, és a szálak mindig konzisztens állapotot látnak
- a példákban a kommenten belül található lépések futnak le egy-egy műveletben:

C#: System.Threadingnamespace

```
// ++counter;
Interlocked.Increment(refcounter);
// tmp= obj; obj= value; return tmp;
tmp= Interlocked.Exchange(refobj, value);
// tmp= obj; if (obj== coparand) { obj= value; } return tmp;
tmp= Interlocked.CompareExchange(refobj, value, comparand);
```

Java: java.util.concurrent.atomicpackage

```
// ++counter;
AtomicIntegercounter= newAtomicInteger(0);
```

```
counter.incrementAndGet();
// tmp= obj; obj= value; return tmp;
AtomicReference<Object> obj= newAtomicReference<Object>();
tmp= obj.set(value);
// if (obj== coparand) { obj= value; return true; } else {
return false; }
obj.compareAndSet(comparand, value);
```

Scoped locking

- amennyiben nincs olyan processzorutasítás, amely adott műveletek csoportját atomiként tudna elvégezni, akkor ezeket a műveleteket egy kritikus szakaszban lehet csoportosítani, amely garantálja a szálak közötti kölcsönös kizárást, így ezek a műveletek a szálak szempontjából atominak tűnnek
- C#-ban egy ilyen kritikus szakaszt a lock kulcsszóval, Java-ban a synchronized kulcsszóval hozhatunk létre

```
lock(obj){ /*...*/ }
synchronized (obj){ /*...*/ }
```

- a kritikus szakaszokkal azonban óvatosan kell bánni
- mivel ezek a szálak között kölcsönös kizárást biztosítanak, a szálak ilyen blokkoknál feltorlódhatnak, hiszen nem tudnak párhuzamosan futni
- éppen ezért törekedni kell arra, hogy a kritikus szakaszon belül minél gyorsabb műveletek, és minél kevesebb művelet legyen
- vigyázni kell arra is, hogy egy-egy ilyen blokkból ne hívjunk bele ismeretlen kódba, például ne hívjunk eseménykezelőt, vagy olyan virtuális függvényt, amelyet a könyvtárunkon kívül mások is felül tudnak írni
- óvatosan kell bánni a kritikus szakaszokkal azért is, mert ha rossz sorrendben zároljuk az objektumokat, akkor az alkalmazás dead-lock-ba kerülhet
- C# és Java esetén a rekurzióval szerencsére nincsen gond, ugyanaz a szál rekurzióban a saját blokkjába még egyszer beleléphet, de más programnyelvekben és könyvtárakban előfordulhatnak olyan megoldások, ahol a rekurzió is dead-lock-hoz vezet

Szinkronizációs minták második csoportja: Balking

Balking design pattern

- a Balking tervezési minta azonnal visszatér, ha az objektum nem a megfelelő állapotban van
- például akkor, ha a feladat végrehajtását egy másik szál már elindította

```
publicclassExample{
    privatebooljobInProgress= false;
    publicvoidExecuteJob() {
        lock (this) {
            if(jobInProgress) {
                return;
            }
            jobInProgress= true;
        }
```

```
            // Code to execute job goes here
            // ...
            lock (this) {
                jobInProgress= false;
            }
        }
    }
```

Singleton implementálási probléma

- Mielőtt rátérnénk a következő megoldásra, a Double-checked locking-ra vizsgáljunk meg egy érdekes problémát
- tegyük fel, hogy a Singleton tervezési mintát szeretnénk implementálni, azonban fel kell készülnünk arra, hogy az alkalmazásunk többszálú
- ha kódban látható naív módon implementálnánk a Singletont, akkor előforulhatna az az eset, hogy két szál egyszerre vizsgálja meg az if feltételét, mind a kettő az látja, hogy a Singleton még nem létezik, így mind a kettő belefut az if törzsébe, mind a kettő létrehoz egy-egy példát a Singleton-ból, és mindkét szál végül más-más objektumot fog látni
- nem sikerült tehát elérni azt a célt, hogy a Singleton-ból pontosan egy példány keletkezzen

```
public class Singleton {
    private static Singleton singleton = null;
    private Singleton() { }
    public static Singleton GetInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

- a probléma megoldható egy kölcsönös kizárással, ha a műveleteket becsomagoljuk egy lock vagy synchronized blokkba
- egy ilyen lock vagy synchronized blokkba való belépés nagyon drága tud lenni, főleg akkor, ha egy másik szál már benne tartózkodik
- a kölcsönös kizárásra mindössze csak akkor lenne szükség, amikor a szingleton létrejön
- amikor a singleton már létezik, akkor nem lenne már szükség a kritikus szakaszra
- ezen az ötleten alapul a double-checked locking

Double-checked locking

- a dupla ellenőrzésű lockolás arról szól, hogy először megvizsgáljuk, hogy a singleton létezik-e már, és ha igen, akkor egyből visszatérünk vele
- ha nem akkor egy kritikus szakasz segítségével hozzunk létre a singletont, ezzel biztosítva, hogy csak egy példány jöjjön belőle létre
- ez a megoldás sokkal hatékonyabb, mert csak akkor futunk rá a kritikus szakaszra, amennyiben valóban szükséges

```
public class Singleton{
    private static object myLock= new object();
    private static Singleton singleton = null;
```

```
        private Singleton() {}
        public static Singleto nGetInstance(){
            // 1st check
            if(singleton == null){
                lock(myLock){
                    // 2nd (double) check
                    if(singleton == null){
                        singleton = new Singleton();
                    }
                }
            }
            return singleton;
        }
    }
```

Double-checked locking problémája

- sajnos ennek a megoldásnak vannak hátulütői
- ez a megoldás nem működik .NET 1-ben és egyáltalán nem működik Java esetén
- a probléma az, hogy amikor létrehozza a singletont, előbb állítja be a singleton változó értékét az objektumra mutató refernciára, a konstruktor pedig csak ezután fog lefutni
- ezzel az a baj, hogy amíg a konstuktor fut, egy másik szál is jöhet, és elkérheti singletont, aki azt fogja látni, hogy a singleton változónak már van értéke, így visszatér vele, és a szál így egy félkész objektummal fog találkozni, amelynek a konstuktora még le sem futott
- a probléma szerencsére megoldható .NET-ben és Java 5-től felfelé is, hogyha a singleton változót volatile-ként definiáljuk

  ```
  private volatile static Singleton singleton = null;
  ```
- sajnos azonban Java 4 és az alatt ez a megoldás sem működik, mert rosszul volt definiálva a volatile szemantikája
- ugyan ma már nem valószínű, hogy Java 4-et használnánk, a probléma mégis rávilágít arra, hogy a double-checked locking implementálása nagyon veszélyes tud lenni, így, ha más programnyelvekben vagy környezetekben akarnánk implementálni, akkor győződjünk meg róla, hogy az adott környezet biztonságosan lehetővé teszi-e ezt

Double-checked locking elkerülése statikus inicializálással

- a double checked locking elkerülésére .NET-ben és Javaban is használható a statikus inicializálás
- a statikus inicializálás garantáltan atomi művelet, így ennél nem áll fenn az a veszély, hogy a singleton-ból két példány keletkezik, vagy éppen egy félkész példányhoz jutunk
- a megoldás hátránya az, hogy nem lusta inicializálásról van szó, a singleton nem csak akkor jön létre, amikor a GetInstance függvényt meghívjuk, hanem már akkor, amikor a singleton típushoz valahol hozzáférünk
- további hátrány, hogy a statikus inicializátorok sorrendjének futása nem determinisztikus, így nem lehet garantálni azt, hogy egy adott singleton előbb jön létre, mint egy másik
- további hátránya ennek a megoldásnak, hogy csak statikus singleton esetén használható

- amikor egy objektumon belül példányszintű singletont szeretnénk létrehozni, akkor ez nem használható

```
public class Singleton {
    private static readonly Singleton singleton =
        new Singleton();
    private Singleton() {}
    public static Singleton GetInstance() {
        return singleton;
    }
}
```

Double-checked locking elkerülése lusta statikus inicializálással

- néhány előbb említett probléma egy kis csavarral megoldható, hogyha a singletont egy belső osztályban hozzunk létre, mert ennél tudjuk kontrollálni azt, hogy mikor férünk hozzá ehhez a típushoz
- így például megoldható az, hogy a singleton csak a GetInstance függvény meghívásakor jöjjön létre, tehát biztosítható a lusta kiértékelés, és a singleton-ok megfelelő sorrendben való meghívásával a determinisztikus inicializálás is működik
- sajnos arra továbbra sincsen megoldás, hogy ha egy objektumon belül szeretnénk példány szintű singletont létrehozni

```
public class Singleton {
    // Lazy initialization:
    private class Holder {
    public static readonly Singleton singleton =
        new Singleton();
    }
    private Singleton() {}
    public static Singleton GetInstance() {
        return Holder.singleton;
    }
}
```

Guarded suspension

- az őrfeltétellel való felfüggesztés minta megvárja, míg sikerül belépni a kritikus szakaszba és egyben egy elvárt előfeltétel is teljesül
- az alábbi Java kódban az látjuk, hogy a synchronized blokk elején egy while ciklus egy adott előfeltételre vár, és ha az nem teljesül, akkor a szál futása felfüggesztésre kerül a wait() függvényhívás hatására
- amikor a szál felfüggesztődik, akkor ideiglenesen a kritikus szakaszból is kilép
- amikor egy másik szál meghívja a lenti függvényt, az beléphet ugyanebbe a kritikus szakaszba, teljesítheti az előfeltételt, és a notify() hívás segítségével felébreszthetti az előzőleg felfüggesztett szálat
- a felébresztett szál futása nem fog azonnal folytatódni, egyelőre futásra kész állapotba kerül, és megvárja, amíg újra beléphet a kritikus szakaszba, miután az őt felébresztő szál már kilépett abból
- a felébresztett szál futása a wait() hívás után folytatódik
- a while ciklus fejlécében újra megvizsgálja, hogy valóban teljesül-e az előfeltétel

- ha igen, akkor kilép a ciklusból, és a synchronized blokkon belül végrehajthatja azokat a műveleteket, amelyeknek szükségük volt az előfeltétel teljesülésére

```
public void operationWithPrecondition() {
    synchronized (lock) {
        while (!preCondition) {
            try {
                lock.wait();
            } catch (InterruptedException e) { }
        }
        // ...
    }
}

public void fulfillPrecondition() {
    synchronized (lock) {
        preCondition = true;
        lock.notify();
    }
}
```

- felmerülhet a kérdés, hogy miért van szükség while ciklusra, miért nem elég egy egyszerű if feltétel
- a válasz az, hogy elképzelhető, hogy több szál is várakozik ugyanennek az előfeltételnek a teljesülésére, és egy notifyAll(), vagy egy másik helyről történő notify() hívás őket is felébresztheti, és a wait-ből visszatérve olyan műveleteket hajthatnak végre, amelyek során az előfeltétel ismét hamissá válik
- éppen ezért, amikor egy szál felébred, újra meg kell vizsgálnia, hogy teljesül-e az az előfeltétel, amelyre ő éppen várakozik
- fontos azt is hangsúlyozni, hogy a szál felfüggesztése mindenképpen szükséges, soha ne írjunk üres törzsű while ciklust, mert az aktív várakozásnak minősül, és 100%-on fogja enni a CPU-t

Guarded suspension - FIFO példa

- az alábbi kódrészlet egy olyan FIFO jellegű üzenetsort ábrázol, amely a Guarded suspension használatára mutat példát
- az üzenetsor egyszerre legfeljebb 10 db üzenetet tud tárolni
- az enqueue() függvény egy új üzenetet rak be a sorba, és nála az az előfeltétel, hogy legfeljebb 9 üzenet legyen a sorban
    - ha ez nem teljesül, akkor ő kénytelen várakozni
- a dequeue() függvény egy elemet vesz ki a sorból
    - ennek nyilvánvaló előfeltétele, hogy legalább egy üzenet legyen a sorban
    - ha ez nem teljesül, akkor ő kénytelen várakozni
- az egész rendszer úgy képzelhető el, hogy vannak termelő szálak, amelyek az enqueue() függvényt hívják, és vannak fogyasztó szálak, amelyek a dequeue() függvényt
- ha a termelők teletöltik a sort, akkor kénytelenek lesznek várakozni, hogy egy fogyasztó kivegyen egy elemet, biztosítva ezzel azt az előfeltételt, amire a termelők várakoztak

- hasonlóan, ha a fogyasztók feldolgoztak minden üzenetet és a sor üres, akkor a termelők tudják teljesíteni a fogyasztók előfeltételét, vagyis, hogy legalább egy üzenet legyen a sorban

```
public class Fifo<T> {
    private Object lock = new Object();
    private ArrayList<T> items = new ArrayList<>();
    public void enqueue(T item) {
        synchronized (lock) {
            while (items.size() > 10) {
                try { lock.wait(); }
                catch (InterruptedException e) {}
            }
            items.add(item);
            lock.notifyAll();
        }
    }
    public T dequeue() {
        T result;
        synchronized (lock) {
            while (items.size() == 0) {
                try { lock.wait(); }
                catch (InterruptedException e) {}
            }
            result = items.get(0);
            items.remove(0);
            lock.notifyAll();
        }
        return result;
    }
    //...
}
```

Szinkronizációs minták harmadik csoportja: Jelzések

Monitor object

- a Monitor object két dolgot biztosít egyszerre: a kölcsönös kizárást és a szálak közti jelzéseket
- a Monitor object segítségével egy szál futása felfüggeszthető, és a jelzés arra szolgál, hogy ezt a szálat fel tudjuk ébreszteni
- Monitor object Java esetén
    - Java esetén minden objektum egyben Monitor object is
    - a kölcsönös kizárást úgy biztosítjuk, hogy az objektumra rászinkronizálunk a synchronized kulcsszó segítségével
    - egy szál futása felfüggeszthető egy ilyen blokkon belül, hogyha a szinkronizált objektumon egy wait() függvényt hívunk

    - ennek hatására a szál ideiglenesen kilép a synchronized blokkból, és egy másik szál ugyanarra az objektumra rászinkronizálva küldhet egy jelzést az alvó szálnak az objektumon végrehajtott notify() hívás segítségével, vagy az összes objektumra várakozó szálnak a notifyAll() függvényhívás segítségével
    - a jelzés hatására a várakozó szálak felébrednek, futásra kész állapotba kerülnek, és arra várnak, hogy újra belépjenek a synchronized blokkba
    - de ez csak azután történhet meg, miután az őket értesítő szál már kilépett abból
    - hogyha visszaemlékszünk az előző FIFO példánkra, ott a Guarded suspension megoldás éppen a Monitor object segítségével volt megvalósítva
- Monitor object C# esetén
    - C# illetve .NET esetén a Monitor object műveletei egy külön statikus osztályba vannak kiszervezve
    - itt tehát a Java-val ellentétben az objektumok publikus interfésze nincs teleszemetelve a Monitor object függvényeivel
    - .NET-ben a Monitor osztály statikus Enter(object) és Exit(object) függvénye biztosítja a kölcsönös kizárást
    - paraméterként meg kell kapniuk, hogy melyik objektumra szeretnénk rászinkronizálni, és C#-ban a lock kulcsszó által jelzett kódblokk a hátérben ilyen függvényhívásokra fordul le
    - egy szálat a Monitor statikus Wait(object) függvényével lehet felfüggeszteni
    - egy szálat a Pulse(object), az objektumra várakozó összes szálat a PulseAll(object) függvénnyel lehet felébreszteni

Semaphore

- a Semaphore egy olyan számláló, amelynek értéke 0 és egy maximum érték között változhat
- a számláló növelése és csökkentése pedig atomi műveletként van megvalósítva
- a Semaphore segítségével egy olyan erőforrás-készlet kezelhető, ahol fixen k db egyforma erőforrásunk van
- a Semapohore maximális értéke ez a k szám, és amint valamelyik szálnak szüksége van az erőforrásra, az csökkenti eggyel a Semaphore számlálóját
    - és ha ez a művelet sikeres, vagyis, a számoló nem nulla értékű volt, akkor a szál beléphet a kritikus szakaszba
    - ha pedig nem sikerült, mert a Semaphore értéke 0 volt, akkor a szál blokkolódik egészen addig, amíg egy másik szál vissza nem adja az erőforrást, vagyis újra meg nem növeli a Semaphore értékét
- a Semaphore úgy is elképzelhető, hogy van k db szobánk, ez a k db erőforrás, van k db egyforma kulcsunk, ezek nyitják a szobákat, a sorban érkező emberek pedig a szálak
- minden ember egy kulcsot kap meg, azzal bemehet egy nem foglalt szobába, és bezárhatja azt
- ha az összes szoba foglalt, vagyis nincsen több szabad kulcs, a Semaphore értéke 0, akkor a következő embernek várakoznia kell egészen addig, amíg valaki ki nem jön a szobából, és vissza nem adja a kulcsot, vagyis, amíg a Semaphore értéke eggyel meg nem nő
- C#-ban és Java-ban is a Semaphore osztály segítségével lehet megvalósítani a Semaphore-t

- a kódrészletek arra mutatnak példát, hogy hogyan lehet létrehozni egy Semaphore-t, valamint, hogy hogyan lehet csökkenteni, illetve növelni a számlálóját
- C#: System.Threading.Semaphore

```
Semaphore semaphore = new Semaphore(initialCount,
maximumCount);
// Decrease counter:
semaphore.WaitOne();
// Increase counter:
semaphore.Release();
```

- Java: java.util.concurrent.Semaphore

```
Semaphore semaphore = new Semaphore(MAX_COUNT);
// Decrease counter:
semaphore.acquire();
// Increase counter:
semaphore.release();
```

Mutex

- a Mutex (Mutual exclusion) jelentése kölcsönös kizárás
- a Mutex segítségével egyetlen erőforráshoz való hozzáférés szabályozható
- a Mutex valójában úgy működik, mint egy olyan Semaphore, amelynek számlálója 1 értékű
- C#-ban erre van külön dedikált osztály is

```
Mutex mutex = new Mutex();
// Acquire:
mutex.WaitOne();
// Release:
mutex.ReleaseMutex();
```

- Java-ban nincs ilyen, ott egy olyan Semaphore-t kell létrehozni, amelynek számlálója ténylegesen 1 értékű

Manual reset event

- ennek a jelzésnek a célja, hogy engedélyezze több szál futását, miután egy művelet eredménye elkészült
- a szálak nyilvánvalóan ennek a műveletnek az eredményére várnak
- a jelzésnek két állapota van: vagy jelez, vagy nem jelez
- jelző állapotban a szálak tovább futhatnak, nem jelző állapotban a szálak blokkolva várakoznak
- a két állapot között manuálisan, explicit függvényhívásokkal kell váltani
- a ManuelResetEvent úgy képzelhető el, mint egy ajtó, amely, ha nyitva van, akkor jelez, és szabad az áthaladás, ha zárva van, akkor nem jelez, és nem lehet átmenni
- az ajtót kézzel, manuálisan kell nyitni, illetve zárni
- .NET-ben a ManuelResetEvent osztály valósítja meg ezt a jelzést
    - a Set() függvényével jelző állapotba, a Reset() függvényével nem jelző állapotba lehet átvinni
    - a szálak a WaitOne() függvényhívással várakoznak rá, amely azonnal visszatér, ha a jelző jelzett állapotban van, és a szál folytathatja a futását

    - a WaitOne() függvény azonban blokkol, ha a jelző nem jelzett állapotban van, és ilyenkor a szálnak egészen addig várakoznia kell, amíg valaki meg nem hívja a Set() függvényt
- Java esetén nincs olyan beépített osztály, amely ezt a működést támogatná
    - de ha szükségünk van rá, akkor könnyen implementálhatjuk

Manual reset event implemetnálása

- a teljes osztály áttekintése:

```
public class ManualResetEvent {
    private final Object monitor = new Object();
    private volatile boolean signaled = false;
    public ManualResetEvent(boolean signaled) {
        this.signaled = signaled;
    }
    public void set() {...}
    public void reset() {...}
    public void waitOne() {...}
    public boolean waitOne(long timeout) {...}
}
```

- set() és reset() metódusok
    - a set() és reset() műveleteket monitor object segítségével lehet implementálni
    - a set() beállítja jelzettre az állapotot, és felébreszti a várakozó szálakat
    - a reset() pedig visszaállítja nem jelzettre az állapotot

```
public void set() {
    synchronized (monitor) {
        signaled = true;
        monitor.notifyAll();
    }
}

public void reset() {
    synchronized (monitor) { //required only in Java 4-
        signaled = false;
    }
}
```

- waitOne() metódus
    - a waitOne() metódus is a minitor object segítségével van implementálva
    - láthatjuk azt is, hogy a guarded suspension minta segítségével várakozunk arra, hogy a jelző jelzett állapotba kerüljön
    - ha ez nem sikerül, akkor felfüggesztjük a szál futását a wait() segítségével, és egészen addig felfüggesztett állapotban maradunk, míg valaki meg nem hívja a set() függvényt, és a notifyAll() segítségével fel nem ébreszt minket

```
public void waitOne() {
    synchronized (monitor) {
        while (!signaled) {
            try {
                monitor.wait();
            } catch (InterruptedException e) {}
```

```
            }
        }
    }
```

- waitOne(long timeout) metódus
    - a waitOne() timeoutos változatában megadhatjuk, hogy meddig szeretnénk várakozni, hogy a jelző jelzett értékre váltson

```
public boolean waitOne(long timeout) {
    synchronized (monitor) {
        long t = System.currentTimeMillis();
        while (!signaled) {
            try {
                monitor.wait(timeout);
            } catch (InterruptedException e) {}
            // Check for timeout
            if (System.currentTimeMillis()-t
                    >= timeout) {
                break;
            }
        }
        return signaled;
    }
}
```

Manual reset event használata példa

- tegyük fel, hogy az első szál feladata, hogy letöltsön valamilyen fájlt
- a többi szál arra várakozik, hogy ez a fájl helyileg elérhető legyen
- ezek a szálak a waitOne()-al addig blokkolnak, amíg az első szál meg nem hívja a letöltés végén a Set() függvényt
- ezután az összes letöltésre várakozó szál tovább folytathatja a futását, és végrehajthatja azt a műveletet, amelynek szüksége volt a letöltött fájlra

```
// Thread 1:
public class Downloader {
    public ManualResetEvent Downloaded { get; }
    public Downloader() {
        this.Downloaded = new ManualResetEvent(false);
    }
    public void Download() {
        this.Downloaded.Reset();
        // ... looong operation ...
        this.Downloaded.Set();
    }
}

// Thread 2:
public class FileOpener {
    private Downloader downloader;
    public FileOpener(Downloader downloader) {
        this.downloader = downloader;
    }
```

```
        public void OpenFile() {
            this.downloader.Downloaded.WaitOne();
            // ... open downloaded file ...
        }
    }
```

Auto reset event

- működésében nagyon hasonló a manual reset event-hez, az egyetlen különbség az, hogy a waitOne() függvény sikeres lefutásának hatására a jelző automatikusan visszaesik nem jelzett állapotba
- szemléletesen az a különbség a manual reset event-hez képest, hogy az egy olyan ajtót reprezentál, amelyet manuálisan kell nyitni és csukni, az auto reset event ezzel szemben olyan, mint egy sorompó, ez is manuálisan nyitható és csukható, de ha valaki áthaladt, akkor automatikusan lecsukódik
- az auto reset event tehát a várakozó szálak közül csak egyetlen egyet enged tovább, miután annak a műveletnek az eredménye rendelkezésre áll, amelyre a szálak várakoznak
- .NET-ben az AutoResetEvent osztály valósítja meg ezt a működést
- Java-ban nincs ilyen beépített osztály, de könnyen implementálható, ahogy az a következőkben látható

Auto reset event implemetnálása

- a teljes osztály áttekintése:

```
public class AutoResetEvent {
    private final Object monitor = new Object();
    private volatile boolean signaled = false;
    public AutoResetEvent(boolean signaled) {
        this.signaled = signaled;
    }
    public void set() {...}
    public void reset() {...}
    public void waitOne() {...}
    public boolean waitOne(long timeout) {...}
}
```

- set() és reset() metódusok:
    - a műlödés eddig ugyanaz, mint manual reset event esetén

```
public void set() {
    synchronized (monitor) {
        signaled = true;
        monitor.notifyAll();
    }
}
public void reset() {
    synchronized (monitor) { //required only in Java 4-
        signaled = false;
    }
}
```

- waitOne() metódus:

- a manual reset event-től való különbség a waitOne() függvényben látható, amelynek utolsó sora automatikusan visszaejti a jelzőt nem jelzett állapotba
- így amikor az utánunk következő szál is felébred, az zárt sorompóval fog találkozni, és visszamegy aludni egészen addig, amíg valaki újra meg nem hívja a set() függvényt

```
public void waitOne() {
    synchronized (monitor) {
        while (!signaled) {
            try {
                monitor.wait();
            } catch (InterruptedException e) {}
        }
        signaled = false;
    }
}
```

- waitOne(long timeout) metódus:
  - itt az auto reset event waitOne() függvényének timeout-os változata látható
  - ez is automatikusan visszaejti a jelzőt nem jelzett állapotba

```
public boolean waitOne(long timeout) {
      synchronized (monitor) {
          try {
              long t = System.currentTimeMillis();
              while (!signaled) {
                  try {
                      monitor.wait(timeout);
                  } catch (InterruptedException e) {}
                  // Check for timeout
                  if (System.currentTimeMillis() – t
                          >=timeout){
                      break;
                  }
              }
              return signaled;
          } finally {
          signaled = false;
          }
      }
  }
```

Auto reset event használata példa

- itt látható egy példa, hogy mire is jó az auto reset event
- tegyük fel, hogy készítenünk kell egy nyomtató drivert, amely egyszerre csak egy Job-ot tud kinyomtatni
- a printerGuard változó egy auto reset event, amely true, vagyis jelzett állapotban indul
- a Print() függvényt hívogathatják a szálak, beadva neki a nyomtatandó Job-ot

- az első szál tovább jut a WaitOne() függvényhíváson, hiszen jelzett állapotban van a printerGuard változó, amelynek értéke automatikusan vissza is esik nem jelzett állapotba
- így a következő szál már blokkolódni fog a waitOne() függvény hívásakor
- ez a blokkolódás egészen addig fennáll, amíg az első szál nyomtat, és amint az végzett, és meghívta a set() függvényt, jelzettre állítva a printerGuard változót, a nyomtatásra várakozó szálak közül egy folytathatja a futását, és kinyomtathatja a saját dokumentumát
- fontos még megjegyezni, hogy mind a Manual reset event mind az auto reset event csak kényelmi funkciókat látnak el, működésük helyettesíthető monitor object-el és kölcsönös kizárással, de az általában egy bonyolultabb, segédváltozókon alapuló megoldáshoz vezetne

```
public class PrinterSpooler {
    private AutoResetEvent printerGuard;
    public PrinterSpooler() {
        printerGuard = new AutoResetEvent(true);
    }
    public void Print(PrintJob printJob) {
        this.printerGuard.WaitOne();
        // If we reach here, we have sole access to the
        printer.
        // ... print the job
        this.printerGuard.Set();
    }
}
```

Readers-writer lock

- ennek segítségével egyetlen erőforráshoz lehet hozzáférést biztosítani, méghozzá úgy, hogy az erőforrást egyszerre többen is tudják olvasni, de egyszerre csak egyvalaki tudja írni, és amíg az írás folyamatban van, addig olvasni nem lehet
- a jelző segítségével minden egyes szál megadhatja, hogy ő olvasási vagy írási jogot szeretne
- és amint valaki írási jogot kér, az éppen aktuálisan olvasó szálak még befejezhetik a futásukat, ezután az író szál megkapja a jogot az erőforráshoz, és amíg ő dolgozik a következő olvasási kérések felfüggesztésre kerülnek
- amint az író végzett, és visszaadta az erőforrást, újra szabad a pálya, és ismét lehet újra több szálon olvasni az erőforrást
- ezzel a fajta jelzővel azonban óvatosan kell bánni, mert ha rosszul használjuk, az könnyen deadlock-hoz vezethet
- .NET-ben a ReaderWriterLockSlim, Java-ban a ReentrantReadWriteLock osztály valósítja meg ennek a jelzőnek a működését

Szinkronizációs minták negyedik csoportja: Publikus intefész

Strategized locking

- a Strategized locking célja, hogy az alkalmazás egyszálú és többszálú működésben is hatékony legyen

- egyszálú esetben ugyanis nincs szükség a kritikus szakaszoknál a drága zárolási műveletekre
- külön kódot fenntartani az egyszálú és a többszálú alkalmazásnak túlságosan nagy overhead-el jár
- a megoldás az, hogy az előbb tanult jelzéseket, mint például a monitor object-eket a Strategy tervezési minta segítségével dinamikusan cserélhetővé tesszük, és implementálunk belőlük egy nullobject változatot is, amely a zárolásokat és feloldásokat, a blokkolásokat üres műveletként implementálja
- a nullobject változatok tehát nem csinálnak semmit, szerepük csupán annyi, hogy az egyszálú esetekben behelyettesíthetőek, és kívülről úgy használhatók, mintha rendes, jelzés jellegű objektumok lennének
- ezáltal az egyszálú esetben hatékonyabb lesz a működés, mert nem kell feleslegesen a drága zárolási műveleteket végrehajtani

Thread-safe interface

- a másik mintánk a szálbiztos interfész, amely azt a problémát hivatott megoldani, hogy egyes programnyelvekben és könyvtárakban hogyha a jelzéseket rosszul használjuk, akkor a rekurzív függvényhívások deadlock-hoz vezethetnek
- a megoldás az, hogy zárolást csak az osztályok publikus függvényei végezhetnek, amelyek aztán tovább hívnak a belső protected illetve privát függvényekhez, és ezek a belső függvények már nem végeznek több szálkezelő műveletet, így működhetnek nyugodtan rekurzívan is

Kontextus minták

Globális kontextus

- ennek célja az, hogy a programkód bármely pontján elérhető legyen valamilyen globális információ anélkül, hogy azt folyamatosan paraméterként kelljen átpasszolgatni
- ez a minta akkor is hasznos lehet, ha a futási környezetünk nem támogat dependency injection-t
- fontos megjegyezni, hogy a globális kontextus főleg egyszálú alkalmazásoknál használható
- többszálú esetben ugyanis, a nem a főszálon futó függvények a scope-on kívül is futhatnak, így nem biztos, hogy van hozzáférésük a globális információhoz
- ez a probléma azonban könnyen megoldható, hogyha szálanként definiáljuk a globális kontextust
- hamarosan meglátjuk, hogy ezt hogyan lehet megtenni
- ehhez azonban szükségünk van egy olyan tárra, amely szálanként eltérő információt tud biztosítani (erre fog szolgálni a Thread-local storage)

Globális kontextus példa

- az alábbi kódrészlet arra mutat példát, hogy a globális kontextus hogyan használható
- a Main függvényben definiálunk egy scope-ot, amelyen belül a globális információ elérhető
- a példában ez az információ a "Hello" szöveg

- a scope-on belül meghívva a SomeMethod() függvényt az az aktuális globális kontextusból ki tudja nyerni a globális scope-ban definiált értéket
- a példából jól kivehető, hogy a globális információt nem kellett átadni a függvénynek, mégis könnyen hozzáfért ahhoz
- ez a megoldás sok helyen hasznos lehet, például egy ilyen globális kontextus hozzáférést biztosíthat az adatbázis kapcsolathoz
- így az adatbázishoz bármelyik függvény könnyedén hozzáférhet anélkül, hogy az adatbázis kapcsolatot paraméterként át kellene adni neki, vagy valamilyen módon eljuttatni a függvényt tartalmazó osztályhoz

```
public class SomeClass {
    public void SomeMethod() {
        // Access the value stored in GlobalContext:
        "Hello"
        string currentValue =
            GlobalContext.Current.SomeValue;
    }
}

public class SomeProgram {
    public static void Main(string[] args) {
        // Make GlobalContext available only in this scope:
        using (var scope =new GlobalContextScope("Hello")){
            SomeClass cls = new SomeClass();
            cls.SomeMethod();
        }
    }
}
```

GlobalContext és GlobalContextScope implementáció

- a globális kontextus implementációja az alábbi ábrán látható
- maga a kontextus osztály csupán adattárolást biztosít
- az inicializálást a scope osztály fogja végezni

```
public class GlobalContext {
    // Instantiated only by GlobalContextScope:
    internal GlobalContext(string value) {
        this.SomeValue = value;
    }
    // An option/value available in the context:
    public string SomeValue { get; }
    // Gets the current context, set only by
    GlobalContextScope:
    public static GlobalContext Current { get; internal
        set; }
}
```

- a GlobalContextScope osztály a konstruktorában kitölti a globális kontextus értékét, a Dispose() függvényben pedig felszabadítja azt
- a globális kontextusban tehát addig él az információ, amíg a scope objektum létezik

```
public class GlobalContextScope : IDisposable {
```

```
        private static object lockObj = new object();
        // Set the GlobalContext if it is not yet set:
        public GlobalContextScope(string value) {
            lock (lockObj) {
                if (GlobalContext.Current != null) {
                    throw new InvalidOperationException(
                    "The global context is already set.");
                }
                GlobalContext.Current = new
                GlobalContext(value);
            }
        }
        // Remove the GlobalContext if the scope is ended:
        public void Dispose() {
            lock(lockObj) {
                GlobalContext.Current = null;
            }
        }
    }
```

Thread-local storage

- a globális kontextus többszálú felhasználásához szükségünk lesz egy olyan tárra, amely szálanként eltérő információt tud biztosítani
- ezt .NET-ben és Java-ban is a ThreadLocal osztály valósítja meg
- az objektum orientáltságnál megtanultuk, hogy egy statikus változó mindig az osztályhoz kapcsolódik
- ennek köszönhetően globálisan mindig egy érték tartozik hozzá, rajta keresztül az összes objektumpéldány ugyanazt az egy értéket látja
- amennyiben egy statikus változó ThreadLocal értéket tárol, akkor a hagyományos, statikus működéshez hasonlóan, ugyanazon a szálon az összes objektum ugyanazt az egy értéket fogja látni
- ha azonban az objektumok különböző szálakról néznek rá a statikus változóra, akkor szálanként eltérő értéket fognak látni
- innen is származik a minta neve: szálanként lokális tároló

Thread-local context

- a szálankénti lokális kontextus hasonló megoldás, mint a globális kontextus azzal a különbséggel, hogy a scope-ok és a contextus-ok is szálanként egyediek
- olyan, mintha minden egyes szál saját kontextussal rendelkezne, de nem látnak át egy másik szál kontextusára

ThreadLocalContext és ThreadLocalContextScope implementáció

- hasonló a GlobalContext implementációjához, azzal a különbséggel, hogy az aktuális kontextust egy ThreadLocal változó tárolja

```
    public class ThreadLocalContext {
        // Instantiated only by ThreadLocalContextScope:
        internal ThreadLocalContext(string value) {
```

```
            this.SomeValue = value;
        }
        // An option/value available in the context:
        public string SomeValue { get; }
        private static ThreadLocal<ThreadLocalContext> current
            = new ThreadLocal<ThreadLocalContext>();
        // Gets the current context, set only by
            ThreadLocalContextScope:
        public static ThreadLocalContext Current {
            get {
                return ThreadLocalContext.current.Value;
            }
            internal set {
                ThreadLocalContext.current.Value = value;
            }
        }
    }
```

- hasonló a GlobalContextScope implementációjához
- ő inicializálja a kontextus értékét és ő is szabadítja fel, és ez a kontextus csak azon a szálon érhető el, amelyen a scope-ot létrehozták
- ennél a scope-nál nincsen szükség kölcsönös kizárásra sem, hiszen más szál nem férhet hozzá ehhez a scope-hoz és ehhez a kontextushoz

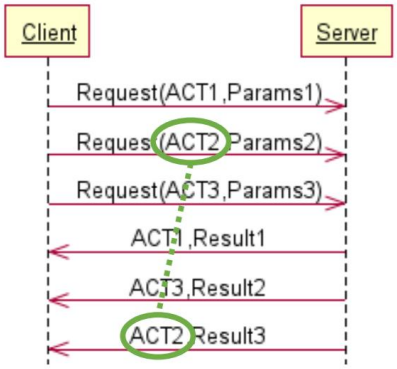
```
    // Create a separate scope for each thread.
    // Locking is not necessary any more,
    // since only the current thread has access,
    // and hence there is no shared state between threads:
    public class ThreadLocalContextScope : IDisposable {
        // Set the ThreadLocalContext if it is not yet set:
        public ThreadLocalContextScope(string value) {
            if (ThreadLocalContext.Current != null) {
                throw new InvalidOperationException( "The
                    thread-local context is already set.");
            }
            ThreadLocalContext.Current = new
                ThreadLocalContext(value);
        }
        // Remove the ThreadLocalContext if the scope is ended:
        public void Dispose() {
            ThreadLocalContext.Current = null;
        }
    }
```

Kérés- és eseménykezelési minták

Asynchronous completion token (ACT)

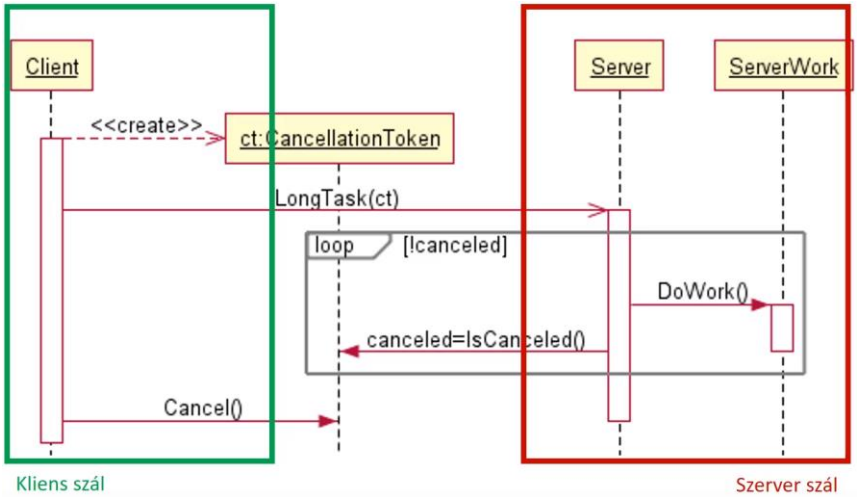
- ez a minta azt a problémát hivatott megoldani, hogy ha a kliens több aszinkron hívást intéz a szerverhez, majd megkapja azokra az aszinkron válaszokat, akkor nehéz eldönteni, hogy melyik kérésre melyik válasz érkezett

- a megoldás az, hogy a kliens minden egyes kérésbe belerak egy azonosítót, ez az asynchronous completion token, amelyet a szerver az egyes válaszokkal együtt visszaküld
- ennek az azonosítónak a segítségével már könnyű összepárosítani, hogy melyik kérésre melyik válasz érkezett



Cancellation token

- ez arról szól, hogyha a kliens elindít egy hosszú ideig futó háttérműveletet, majd később rájön, hogy még sincsen szüksége a művelet eredményére, akkor legyen lehetősége megszakítani a művelet végrehajtását
- ezt úgy lehet elérni, hogy amikor a kliens elindítja a hosszú műveletet, akkor átadja neki paraméterként a cancellation token-t, és a szerver a művelet végrehajtása közben időnként ránéz erre, hogy kezdeményezte-e már a kliens a művelet végrehajtásának visszavonását
- hogyha igen, akkor a szerver abbahagyja a futást és nem végez több munkát
- tipikus példája ennek a helyzetnek az, amikor a programozó gépel a fejlesztőkörnyezetben, a környezet pedig a háttérben folyamatosan fordítja az alkalmazást, de minden egyes billentyűleütés az előző számítást érdektelenné teszi, így a háttérben futó folyamatot meg kell szakítani és egy új számítást kell kezdeményezni



Cancellation token példa

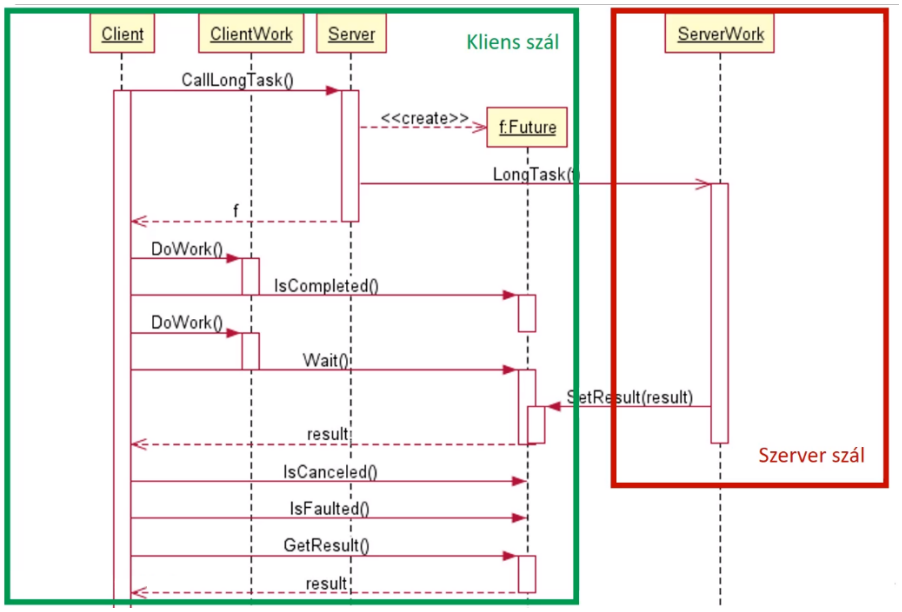
- az alábbi kód bemutatja, hogy C#-ban hogyan működik a cancellation token
- a kliens egy cancellation token source-t hoz létre, ettől kérhető el maga a token, és ezen keresztül tudja a kliens visszavonni a műveletet is
- ezeket láthatjuk a kliens onEdit() függvényében

- a szerver csak a cancellation tokent látja, és időnként rákérdezhet az a isCancellationRequested propery segítségével
- ha a szerver futása éppen a hívási stack mélyében lévő függvényben van, akkor a cancelation tokent meg lehet kérni, hogy dobjon egy kivételt, amelynek segítségével sokkal könnyebb kilépni a hívási stack mélyéről, minthogyha csak a property-t használnánk

```
public class Client {
    private CancellationTokenSource tokenSource =
        new CancellationTokenSource();
    private Server server = new Server();
    public void OnEdit() {
        // Cancel previous compilation
        tokenSource.Cancel();
        tokenSource = new CancellationTokenSource();
        Task.Run(() => server.Compile(tokenSource.Token));
    }
}

public class Server {
    public Task<bool> Compile(CancellationToken token) {
        this.DoWork();
        // Return if the operation is cancelled:
        if (token.IsCancellationRequested) return
        Task.FromResult(false);
        this.DoMoreWork(token);
        return Task.FromResult(true);
    }
    private void DoMoreWork(CancellationToken token) {
        this.DoWork();
        // Return from a deep call by throwing
            OperationCanceledException
        // if the operation is canceled:
        token.ThrowIfCancellationRequested();
        this.DoWork();
    }
}
```

Future/Task/Deferred

- ennek lényege az, hogy a kliens szinkron elindít a háttérben egy hosszú számítást, majd előbb-utóbb értesülni szeretne annak eredményéről, vagy éppen be szeretné várni az eredmény elkészültét
- a future vagy task objektumot az aszinkron művelet adja vissza eredményként, és ez az aszinkron művelet végrehajtásáról egy csak olvasható képet ad
- le lehet tőle kérdezni, hogy fut-e még művelet, végzett-e már, vagy esetleg valaki visszavonta-e
- segítségével a kliens lock-olva is bevárhatja amíg a művelet elkészül



Future/Task/Deferred .NET példa

- a .NET-ben task-nak hívják ezt a megoldást, és az alábbi kód arra mutat példát, hogy C#-ban milyen függvényeken keresztül lehet elérni a task egyes állapotait

```
public void CallLongTaskAndDoSomeWork() {
    CancellationTokenSource ct =
        new CancellationTokenSource();
    Task<CalculationResult> task =
        this.CallLongTask(ct.Token);
    this.DoWork();
    // Check if the task is completed
    if (task.IsCompleted) { }
    this.DoWork();
    // Wait for the task to finish
    task.Wait();
    // Get the result
    CalculationResult result = task.Result;
    // Check if the task is canceled
    if (task.IsCanceled) { }
    // Check if the task has thrown an exception
    if (task.IsFaulted) { }
}

public Task<CalculationResult>
CallLongTask(CancellationToken token) {
    // Cancellation token is optional,
    // included only for the sake of this example:
    Task<CalculationResult> task =
        new Task<CalculationResult>(this.LongTask, token);
    // Start the task in the background
    task.Start();
    return task;
}
```

```
public CalculationResult LongTask() {
    CalculationResult result = new CalculationResult();
    this.DoWork();
    return result;
}
```

Future/Task/Deferred Java példa

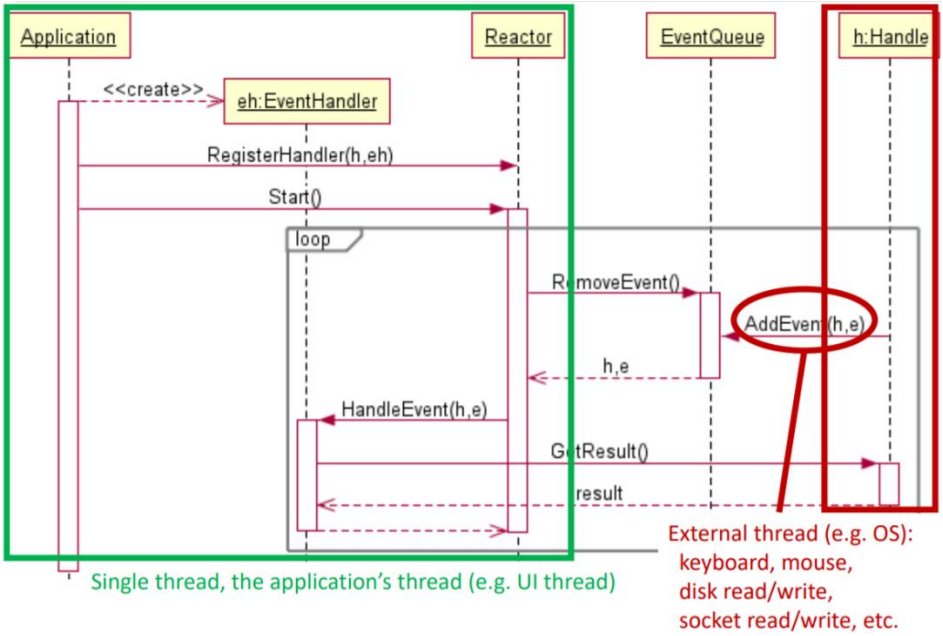
- Java esetén a megoldásnak future a neve, az alábbi kódrészlet pedig ennek a használatára mutat példát

```
private static final ExecutorService threadpool =
    Executors.newFixedThreadPool(3);
public void callLongTaskAndDoSomeWork() {
    Future<CalculationResult> future = this.callLongTask();
    this.doWork();
    // Check if the task is completed
    if (future.isDone()) { }
    this.doWork();
    // Cancel the task
    future.cancel(true);
    this.doWork();
    // Wait for the task to finish
    CalculationResult result = future.get();
    // Check if the task is canceled
    if (future.isCancelled()) { }
    // To check exceptions: override FutureTask
    // or wrap the task to catch and store exceptions
}

public Future<CalculationResult> callLongTask() {
    FutureTask<CalculationResult> task =
        new FutureTask<CalculationResult>(this::longTask);
    // Start the task in the background
    threadpool.execute(task);
    return task;
}

public CalculationResult longTask() {
    CalculationResult result = new CalculationResult();
    this.doWork();
    return result;
}
```

Reactor

- a Reactor nem blokkoló, szinkron eseményfeldolgozást tesz lehetővé
- ez azt jelenti, hogy az események kívülről aszinkron módon érkeznek, vagyis az eseményt küldő kliens nem blokkolódik
- az esemény feldolgozása pedig egy szálon, sorban egymás után történik, tehát az eseményfeldolgozók blokkolják egymást

- a minta szereplői
    - a Handle az az entitás, aki az eseményt generálja
    - ez lehet például az operációs rendszer, amely billentyűzet eseményt vagy egér eseményt generál, de lehet például a szerverünk kliense, amely http kérést intézett hozzánk, és ez generálta az eseményt
    - az események az EventQueue-ban, vagyis az eseménysorba kerülnek be
    - a Reactor olyan ciklust futtat, amely folyamatosan eseményeket vesz ki az eseménysorból
    - az EventHandler tartalmazza azt a kódot, amelyet egy adott eseményre való reakcióként le kell futtatni
    - az alkalmazás az eseménykezelőket beregisztrálja a Reactornál, megadva, hogy mely eseménykezelőt mely eseménynél kell lefuttatni
    - majd az alkalmazás elindítja a Rector fő ciklusát
    - a Reactor a fő ciklusában egyenként veszi ki az eseményeket az eseménysorból, majd keresi és lefuttatja a hozzájuk beregisztrált eseménykezelőt



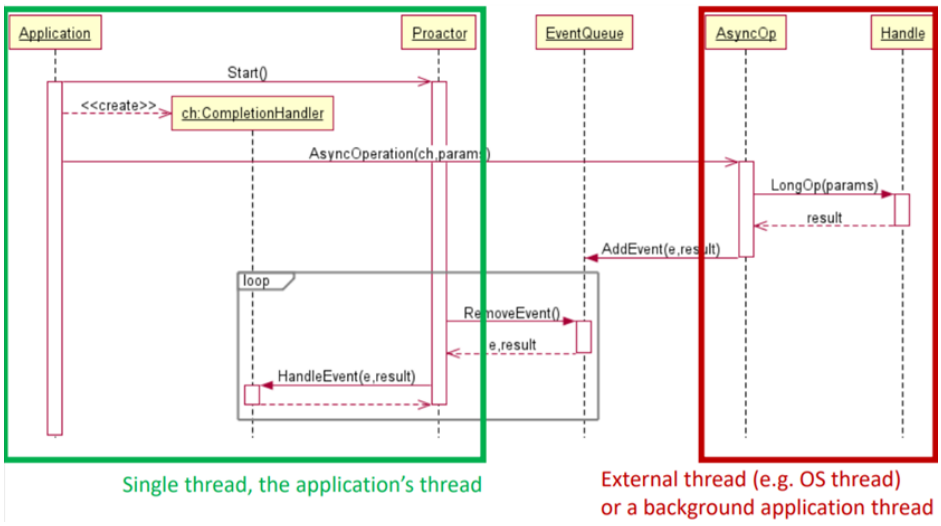
- a minta szálkezelése
    - az alkalmazás és a reaktor mind ugyanazon az egy szálon futnak
        - .NET-ben ez a UI Thread
    - a NodeJs JavaSctript alkalmazásszerver esetén pedig a fő szál, ami egyben az alkalmazás egyetlen szála is
    - az eseményeket generáló Handle egy másik szálon fut
        - tipikusan ez az operációs rendszer, amely mindenféle eseményeket generál
    - fontos látni tehát, hogy a kívülről érkező események nem blokkolják egymást, azonban az eseményekre adott reakciók, az EventHandler-ek ugyanazon az egy szálon egymás után futnak le, és így blokkolják egymást
    - biztosan mindenki találkozott már azzal az esettel, amikor az operációs rendszer azt írja ki, hogy egy alkalmazás nem válaszol
        - ez tipikusan annak a jele, hogy egy eseménykezelő még nem lépett ki, így blokkolja a többi esemény végrehajtását, vagyis az alkalmazás nem

tud reagálni sem az egéreseményekre, sem a billentyűzet eseményekre, sem pedig az ablak újra rajzolását célzó eseményekre sem

- fontos tehát, hogy olyan eseménykezelőket írjunk, amelyek gyorsan lefutnak, hogy ne blokkoljuk el a többi esemény lekezelését
- hogyha olyan műveletet kell megoldanunk, amely sokáig tart, akkor arra indítsunk egy külön szálat az eseménykezelőből, hogy ne blokkoljuk vele a Reactor ciklusát
- az is fontos, hogy az eseménykezelő ne dobjon kivételt, mert az az egész alkalmazást le fogja állítani

Proactor

- a Proactor lényege, hogy a háttérben aszinkron módon, hosszú lefutású műveleteket kezdeményezünk, majd, amikor egy-egy ilyen művelet végzett, az egy eseményt generál, a Proactor pedig ezekre az eseményekre fog reagálni
- a minta szereplői
  - a Handle az, aki a hosszú lefutású műveletet végrehajtja
  - az AsyncOP aszinkron módon hívható, és ő az, aki a Handle-t futtatja, majd a keletkezett eredményről egy eseményt generál
  - az esemény bekerül egy eseménysorba, amelyet egy Proactor egy cikluson belül folyamatosan olvasgat
  - az alkalmazás indítja el a Proactor eseménykezelő ciklusát
  - és az alkalmazás indítja el az AsyncOp-on keresztül a hosszú lefutású műveletet is
  - a művelethez az alkalmazás egy CompletionHandlert is definiál
  - ezt fogja lefuttatni a Poactor akkor, amikor a művelet eredményéhez tartozó eseményt feldolgozza
  - a CompletionHandler további háttérműveleteket indíthat egy másik AsyncOp-on keresztül
  - az alkalmazás viselkedése tehát a CompletionHandler-ek egymásba láncolásával alakul ki
  - ezt a programozási modellt követi például a NodeJs server



- a minta szálkezelése
  - a Proactor tehát a Reactorhoz hasonlóan az egyetlen szálon fut, ezen az egyetlen szálon dolgozza fel a beérkező eseményeket

- a hosszú lefutású műveleteket a háttérben egy külön szál hajtja végre, de a művelet futhat kívül is, például az operációs rendszerben vagy egy MongoDB adatbázisban
- ahogy a Reactor-nál, itt a Proactor-nál is fontos, hogy ne blokkoljuk el a Proactor esemény-végrehajtási ciklusát
- a CompletionHandler legyen rövid, egyszerű, és hogyha egy hosszú ideig tartó műveletre van szüksége, akkor azt egy aszinkron operáción keresztül kezdeményezze a háttérben

Reactor vs Proactor minta

- a különbség a Reactor és Proactor minta között az, hogy a Reactorban az alkalmazás passzív, arra vár, hogy kívülről eseményeket kapjon, míg a Proactor esetében az alkalmazás aktív, ő kezdeményezi a hosszú lefutási idejű műveleteket a háttérben, és az alkalmazás arra reagál, hogyha a művelet eredményének elkészülte eseményt generál
- egy ilyen eseménykezelő pedig újabb háttérben futó műveleteket indíthat
- ilyen értelemben a JavaScript alapú NodeJS szerver mind a Reactor, mind a Proactor mintát követi
    - a kliensektől beérkező http kéréseket Reactor módon, a háttérben indított műveleteket, például a MongoDB adatbázishoz való hozzáférést pedig Proactor módon dolgozza fel

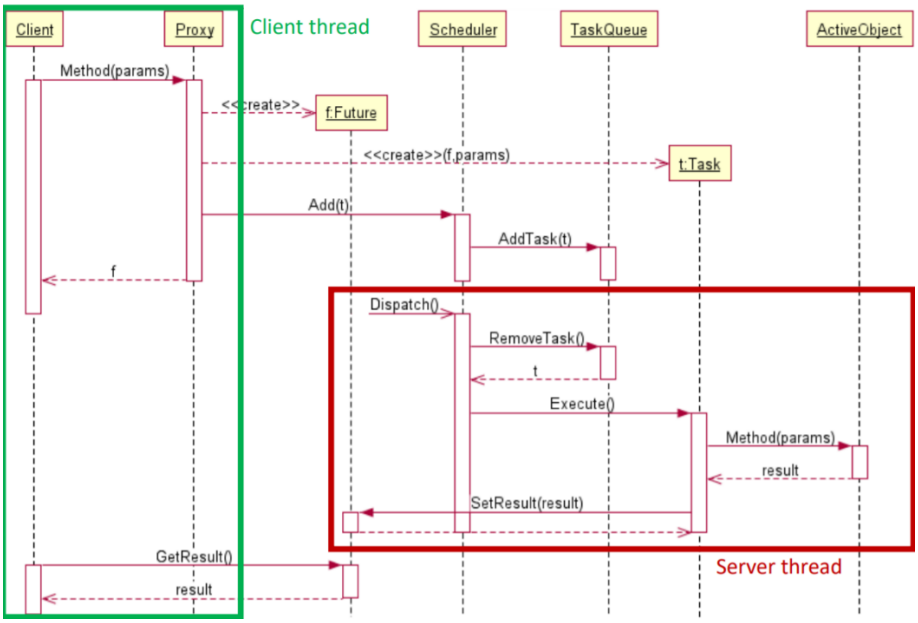
Konkurencia minták

- a konkurencia minták a többszálú feldolgozásról szólnak
- a konkurencia minták közé tartozik a Monitor object is, amiről korábban már kifejtésre került

Active object

- az Active object célja a függvény meghívásának és a függvény lefutásának szétcsatolása
- a hívó kliens objektum és a futtató szerver objektum tehát külön szálon futnak
- az Active Object mintát követik .NET-ben például a Task-ok, Java-ban pedig az ExecutorService
- a minta szereplői
    - szükségünk van egy interfészre, amelyet a szerver objektum, az ActiveObject implementál
    - a kliens által hívott Proxy objektum ennek az interfésznek egy aszinkron változatát implementálja, amely nem közvetlenül a függvény eredményével, hanem egy olyan Future-el tér vissza, amely a függvény eredményét később szolgáltatni tudja
        - ez a future ugyanaz, amit már korábban megbeszéltünk
    - látható, hogy ezek a minták szorosan összefüggnek egymással
    - amikor a kliens a Proxy objektumot meghívja, a Proxy létrehozza a Future-t, ami később az eredményt szolgáltatja a kliensnek, és létrehoz egy Task-ot, ami később a háttérben a függvény végrehajtásáért fog felelni

- a Proxy odaadja ezt a Task-ot egy ütemezőnek, a Scheduler-nek, ami berakja azt egy feladatsorba, a TakQueue-ba
- a Dispatch() hívás időnként beérkezik az ütemezőhöz, ez jelzi, hogy egy újabb feladatot lehet végrehajtani
- az ütemező kiveszi a következő feladatot, és végrehajtja
- a feladat meghívja az ActieveObject megfelelő függvényét, az eredményt pedig beállítja a Future-ben
- a kliens pedig valamikor majd ránéz erre a Future-re, és lekéri az eredményt



- a minta szálkezelése
  - a kliens és a Proxy fut tehát a kliens szálon
  - a Scheduler Dispatch függvénye és a Task, valamint az ActiveObject pedig a szerver szálon
- az Active object minta tehát nagyon hasonlít az elosztott kommunikációra, amikor a kliens és a szerver külön memóriatérben vannak
- ott is szét van csatolva a kliens által történő függvényhívás, és a szerver által történő függvény-végrehajtás
- az Active Object mindössze annyiban más, hogy a kliens és a szerver ugyanabban a memóriatérben vannak, csak különböző szálakon futnak

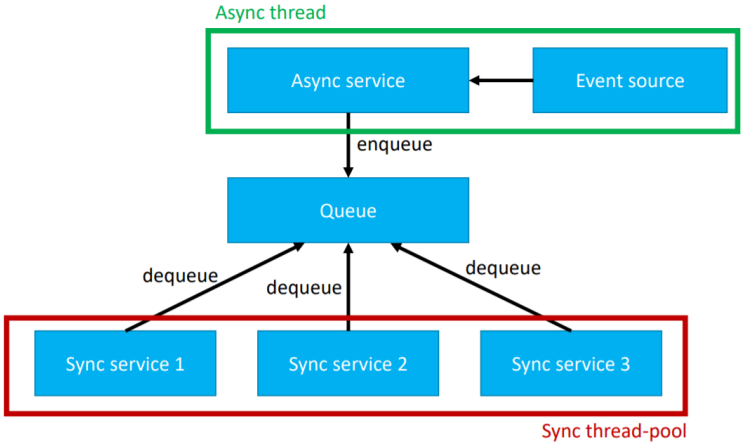
Half-sync/half-async

- a fél szinkron fél aszinkron minta lényege, hogy a szinkron és az aszinkron végrehajtást szétcsatolja egymástól
- a mintának két változata van: az egyik amikor aszinkron kliens hív szinkron szervert, a másik pedig, amikor szinkron kliens hív aszinkron szervert
- a szétcsatolás egy task-okat tartalmazó soron keresztül történik

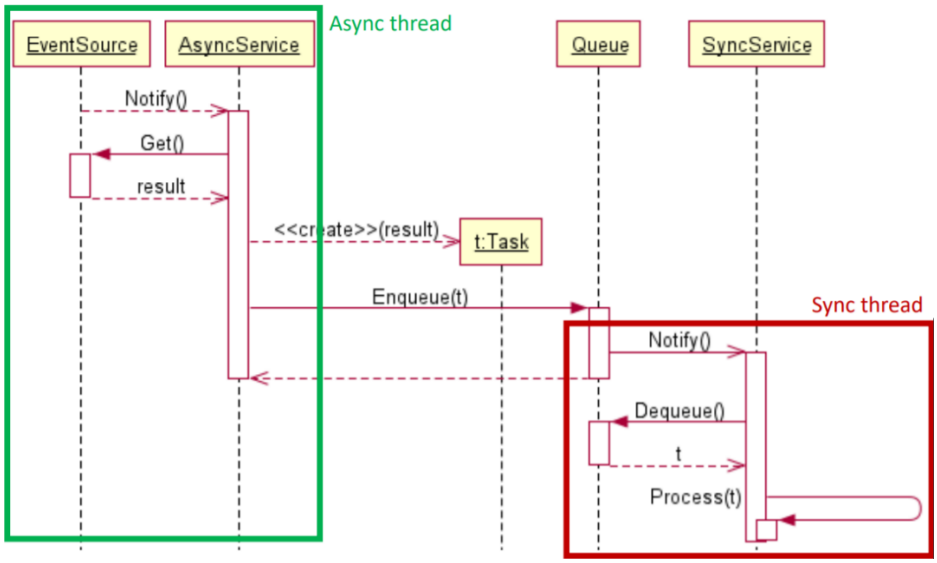
Half-sync/half-async - async->sync

- ebben az esetben egy aszinkron kliens (vagyis aszinkron eseményforrás) hív szinkron szervert
- lényegében tehát aszinkron módon érkeznek be feladatok, és szinkron módon dolgozzuk őket fel

- a feldolgozó szerver objektumokból több is lehet, és ezek külön szálakon futnak



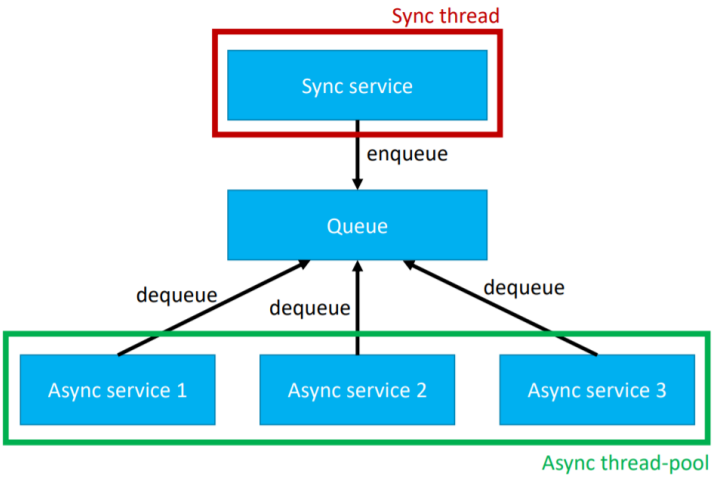
- a minta szereplői
  - az EventSource, vagyis az eseményforrás generálj az aszinkron eseményt
  - ő hívja meg az aszinkron szolgáltatást, az AsyncService-t
  - ez létrehoz egy Task-ot, amelyet elhelyez a feladatokat tároló sorban
  - a sor értesíti a szinkron feldolgozókat, hogy újabb feladat érkezett



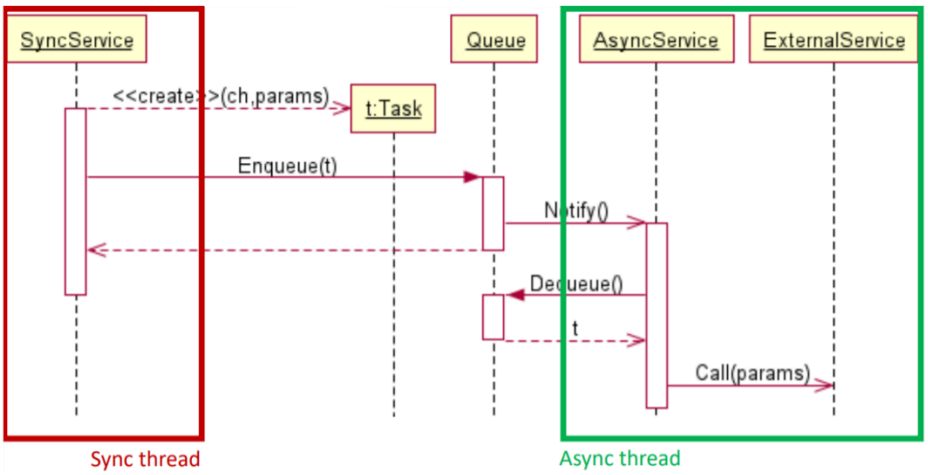
- a minta szálkezelése
  - a szinkron feldolgozók külön szálon futnak, amelyik éppen szabaddá válik, az kiveszi a következő feladatot a sorból, és szinkron módon a saját szálán feldolgozza
  - hogyha alaposan végiggondoljuk, akkor észrevehetjük azt, hogyha a szinkron kiszolgálókból csak egyetlen egy darab van, tehát csak egy szálon fut a kiszolgálás, akkor éppen a Reactor mintát kapjuk vissza

Half-sync/half-async - sync->async

- ebben az esetben a szinkron irányból megyünk az aszinkron irányba
- itt egy darab szinkron szál generálja a feladatokat, berakja őket egy feladatsorba, és az aszinkron feldolgozók ezekből válogatnak, meghívva egy külső szolgáltatást



- a minta szereplői
  - a szinkron szolgáltatás generálja a Task-okat, amelyek egy feladatsorban gyűlnek
  - az aszinkron szerverek értesülnek, ha új feladat érkezik
  - a következő szabad kiszolgáló kiveszi a következő szabad feladatot, majd aszinkron módon végrehajtja azt egy külső kiszolgáló segítségével



- a minta szálkezelése
  - a Task-okat egy szinkron szál készíti szinkron módon
  - a feldolgozás pedig külön szálon, aszinkron módon történik
  - hogyha alaposan végiggondoljuk, akkor a mintának ez a változata pedig a Proactor mintára hasonlít

Leader-followers

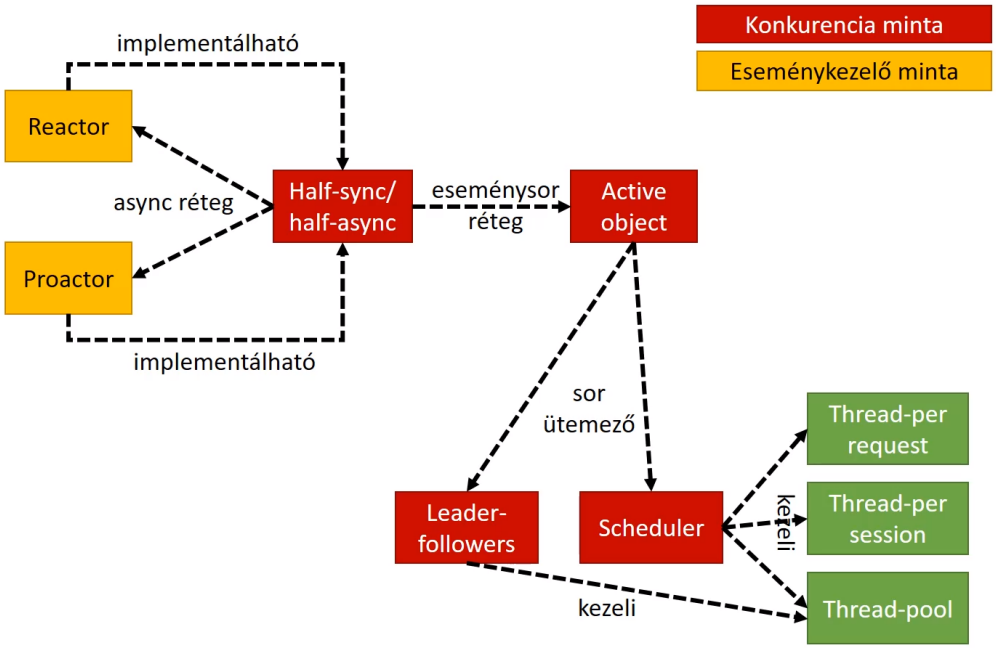
- ez a minta egy nagyon egyszerű ütemezést valósít meg
- a minta lényege, hogy van egy készletünk a szálakból, amelyek közül egy a vezető, a Leader, a többiek a követők, a followers
- a követő szálak mind arra várnak, hogy egyszer majd vezetők lehessenek
- sorra érkeznek a feladatok, és mindig a vezető szál kapja a következőt
- ilyenkor a vezető szál a követő szálak közül egyet előléptet vezetővé, ő pedig elkezd dolgozni a feladaton, és amikor végzett, ő is követővé válik, arra vár, hogy egyszer majd újra vezető lehessen, és újra feladatot kapjon

Scheduler

- egy okosabb ütemezést valósít meg a Scheduler minta
- a Scheduler-hez is folyamatosan érkeznek be a feladatok, tipikusan egy feladatsoron keresztül
- az ütemező pedig több szálat kezel egyszerre tipikusan egy ThreadPool-ból, és valamilyen algoritmus alapján ezeknek osztogatja ki a feladatokat
- az algoritmus sokkal komplexebb is lehet, mint amit a Leader-followers csinál
- figyelembe lehet például venni a feladatok sajátosságait, lehet közöttük akár prioritást is definiálni
- vagyis hogyha van egy dedikált ütemezőnk, amelyben valamilyen komplexebb algoritmust megvalósítható, akkor sokkal rugalmasabb ütemezést tudunk megvalósítani, cserébe azonban fel kell vállalnunk, hogy sokkal bonyolultabb lesz megvalósítani az ütemezést

Kapcsolat a minták között

- láthattuk, hogy a Reactor és a Proactor minta is implementálható a Half-sync/half-async segítségével, de a Half-sync/half-async minta aszinkron rétege is implementálható Reactor illetve Proactor mintával is
- ha ezek így szépen körbeérnek, akkor kapjuk meg például a NodeJS szervert
- a Half-sync/half-async mintában van egy sor, ami a Task-okat tartalmazza, ezeknek a végrehajtását az ActiveObject minta valósítja meg
- azt, hogy melyik legyen a következő megoldandó feladat egy ütemező dönti el
- az ütemező lehet egy egyszerűbb változat, mint a Leader-followers, de lehet akár egy bonyolultabb is, mint a Scheduler
- a Leader-followers szálak egy készletéből dolgozik
- a Scheduler már lehet rugalmasabb, ő is dolgozhat Thread-pool-ból, de az is lehet, hogy kérésenként indít egy új szálat, vagy követi azt, hogy ki a konkrét kliens, és kliensenként, vagyis session-önként tart fenn egy szálat
- jól látható tehát, hogy ezek a minták szorosan összefüggenek egymással


Konkurencia minta
Eseménykezelő minta
implementálható
Reactor
async réteg
Half-sync/
half-async
eseménysor
réteg
Active
object
Proactor
implementálható
sor
ütemező
Thread-per
request
Thread-per
session
Thread-pool
Leader-
followers
Scheduler
kezeli
kezeli

Immutable objektumok

Immutable objektumok

Immutable (nem módosítható) objektumok

- egy objektumot akkor nevezünk immutable-nek, ha az állapotát a létrehozás után már nem tudjuk befolyásolni
- egy ilyen objektumot a konstruktorával inicializálunk, és nincsenek neki setterei, vagy olyan metódusi, amelyek az állapotot módosítani tudnák
- lehetnek azonban lusta kiértékelésű belső attribútumai, amelyek csak akkor kapnak értéket, ha ténylegesen lekérdezzük őket, de ezt az értéket is előre meghatározza az, ahogy az objektumot a konstruktorban létrehoztuk
- ezt az értéket a konstruktoron kívül semmilyen más későbbi függvényhívás nem befolyásolhatja
- nem módosítható, vagyis immutable objektumok például a String osztály példányai Java-ban és .NET-ben is

Immutable objektumok előnye

- az immutable objektumok legnagyobb előnye, hogy szálbiztosak, hiszen belső állapotuk nem módosítható, így bármelyik szálról is nézünk rájuk, mindig ugyanazt látjuk

Immutable objektumok használata

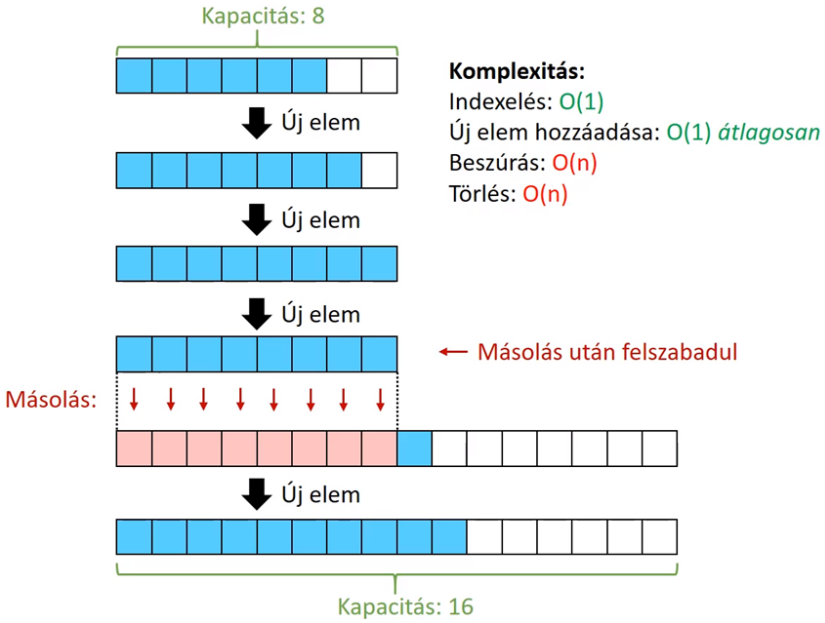
- a kérdés persze felmerül, hogy hogyan lehet úgy programozni, ha az objektumaink nem módosíthatók
- a válasz az, hogy módosítás helyett egy új immutable objektumot kell létre hozni, amelyik már az új állapotot tartalmazza
- ilyenkor kérdésként az is felmerül, hogy nem drága-e az, hogy mindig újabb és újabb objektumokat kell létrehozni
- a válasz pedig az, hogy nem feltétlenül, mert az állapotnak azon része, amelyik nem változik, az újrahasznosítható
- sőt, látni fogjuk, hogy bizonyos esetekben az immutable objektumok még hatékonyabbak, mint a módosítható változataik

Mutable és Immutable implementációk összehasonlítása

Mutable lista tömbbel implementálva: List<T>

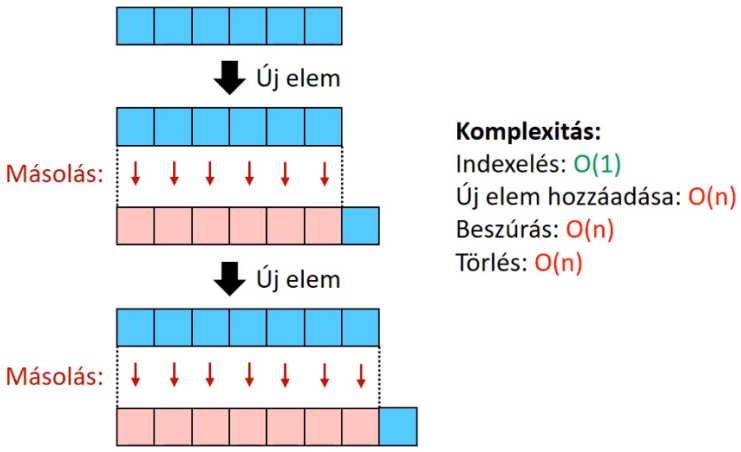
- egy ilyen listát tipikusan úgy implementálunk, hogy az elemeket tároló tömb nagyobb, mint ahány elem ténylegesen van a listában
- ennek tömbnek a méretét kapacitásnak hívjuk
- amikor ez betelik, és még egy új elemet kellene beszúrni, akkor megduplázzuk ennek tömbnek a méretét, a régi elemeket átmásoljuk, és így már lesz hely a következő elemnek, valamint rajta kívül még jó néhány másik elemnek

- ennek a megoldásnak az előnye, hogy bár vannak olyan elemek, ahol a hozzáadás művelet elég drága, hiszen az egész tömb lemásolását igényli, de átlagosan új elemekre vetítve a hozzáadás művelet időigénye konstans
- adott indexű elem lekérése is konstans idejű, hiszen csak bele kell indexelni a tömbbe
- a beszúrás és törlés viszont lineáris, mert a beszúrt vagy törölt elem mögötti elemeket mozgatni kell



Immutable lista tömbbel implementálva

- ha tömb segítségével implementálnánk egy immutable listát, akkor a működés az lenne, hogy a tömböt mindig egyesével növelgetjük, mindig átmásolgatjuk, így egy új elem hozzáadása is lineáris idejű lesz
- a többi művelet komplexitása nem változik

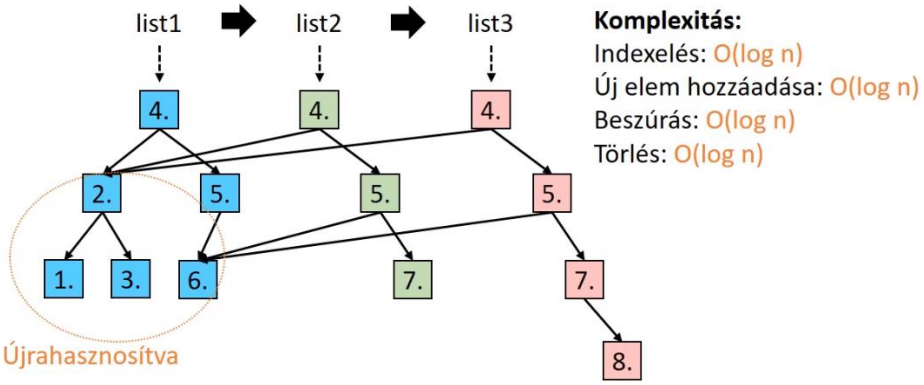


- nyilvánvalóan ez a nem módosítható megoldás rosszabb, mint a módosítható változat
- lehet azonban ezt okosabban is csinálni, ha a lista elemei nem tömbben, hanem egy kiegyensúlyozott bináris fában tároljuk

Immutable lista bináris fával: ImmutableList<T>

- ebben az esetben egy elem hozzáadása mindössze a fának a bővítését jelenti, de a fának azon részei, amelyek nem változnak újrahasznosíthatók

- így egy elem hozzáadásának művelete lineáris komplexitás helyett logaritmikus komplexitásúvá válik
- az indexelés ugyan lassabb lesz (konstans helyett logaritmikus), azonban a beszúrás és törlés művelet is logaritmikussá válik, ami a lineárishoz képest egy nagyon jó eredmény




```
list2 = list1.Add(item)
list3 = list2.Add(item)
```

- látható tehát, hogy az immutable megvalósítás sem feltétlenül lassú, hiszen a nem változó részek újrahasznosíthatók, de hamarosan látni fogjuk, hogy egyes helyzetekben sokkal hatékonyabbak, mint a módosítható megfelelőik

Problémák a módosíthatósággal

- readonly/final csak a referencián keresztüli módosítást tiltja
- módosítható érték átadása paraméterként veszélyes
- módosítható értékkel való visszatérés veszélyes
- többszálú elérés bonyolult
- változó identitás gondot okozhat kulcsoknál
- hiba esetén korrupt belső állapot jöhet létre
- időbeli csatolást hoz létre

Readonly/final

- Java-ban és C#-ban a final illetve a readonly nem garantálják azt, hogy az objektum csak olvasható, mindössze annyit garantálnak, hogy az objektumra mutató referencia nem módosítható
- az alábbi példa azt mutatja, hogy a Zero nevű dátum hiába readonly, maga az objektum még mindig módosítható, így például, ha konstansnak szánjuk a program futása során, fennáll a veszélye annak, hogy egyes függvények elrontják a belső állapotot, és mégsem lesz konstans az objektum értéke

```
public class Date {
    public int Year { get; set; }
    public int Month { get; set; }
    public int Day { get; set; }
}

public class Program {
    private static readonly Date Zero = new Date();
```

```
        public static void Main(string[] args) {
            Zero.Year = 2000; // Zero is still mutable
        }
    }
```

Módosítható érték átadása paraméterként

- a probléma az, hogy semmi nem garantálja azt, hogy a függvény, amit meghívtunk, nem fogja elrontani a paraméterként beadott objektum állapotát

```
    public int Min(List values) {
        values.Sort();
        return values.First();
    }
```

Módosítható értékkel való visszatérés

- ugyancsak veszélyes egy belső állapotot tároló módosítható objektumot kiadni a kezünk közül a külvilágnak
- a külvilág bármikor belenyúlhat és elronthatja a mi állapotunkat is

```
    public class String{
        private char[] chars;
        // ...
        public char[] GetChars(){
            return this.chars;
        }
    }
    public class Program{
        public static void Main(string[] args){
            String str = new String("Hello");
            char[] chars = str.GetChars();
            chars[0] = 'B';
        }
    }
```

- a megoldás ilyenkor az, hogy a belső állapotot nem adjuk vissza közvetlenül, hanem készítünk egy védelmi másolatot, és ezt osztjuk meg a külvilággal
- a másolatot a külvilág nyugodtan manipulálhatja, ez a mi belső állapotunkra nem lesz hatással

```
    public class String{
        private char[] chars;
        // ...
        public char[] GetChars(){
            return (char[])this.chars.Clone();
        }
    }
```

- látszik azonban, hogy meglehetősen nagy árat fizetünk ezért, mert minden egyes lekéréskor el kell készítenünk a másolatot

Többszálú elérés

- a következő probléma abból adódik, hogyha több szálon szeretnénk elérni az objektumot

- ha például két szál egyszerre hívja meg a push függvényt, elképzelhető, hogy ugyanazt a top értéket látják, mind a ketten
- ezt az értéket megnövelve ugyanahhoz az új értékhez jutnak, és az egyik szál felül fogja írni a másik szál által beírt értéket

```
public class Stack<T> {
    private int top = 0;
    private T[] items = new T[100];
    public void Push(T item) {
        items[top++] = item;
    }
    public T Pop() {
        return items[top--];
    }
}
```

- természetesen ez ellen lehet védekezni a tanult párhuzamos és konkurencia mintákkal, azonban ezek sokszor meglehetősen bonyolult programozási modellhez vezetnek

Változó identitás

- következő probléma a változtathatósággal, hogy változik az objektumok identitása
- az objektumok hash-kódja ugyanis a belső állapothoz kötött
- az alábbi példában egy Alice nevű személyt használunk kulcsként a Dictionary-ban, majd a nevét átírjuk Bob-ra, és ezután ezt az objektumot már nem tudjuk kulcsként használni, mert megváltozott a hash-kódja

```
Dictionary map = new Dictionary();
Person p = new Person("Alice");
map.Add(p, "Hello");
p.Name = "Bob";
string value = map[p]; // KeyNotFoundException
```

Hiba esetén korrupt belső állapot

- a következő probléma a módosíthatósággal, hogy a belső állapot inkonzisztensé válhat egy kivétel keletkezése során
- az alábbi kódrészletben például az elemek számát mutató size értéke és a tényleges elemeket tároló items tömb szétcsúszhatnak egymástól, hogyha kivétel keletkezik

```
public class Stack {
    private int size;
    private String[] items;
    // ...
    public void push(String item) {
        size++;
        if (size > items.length) {
            throw new RuntimeException("stack overflow");
        }
        items[size-1] = item;
    }
}
```

- ez a konkrét példa ugyan kicsit mesterkélt, nyilván a size növelését lehetni az után is csinálni, hogy ellenőrizzük a feltételt, de ennél sokkal bonyolultabb esetekben előfordulhat, hogy nem tudjuk garantálni a konzisztenciát
- például, mert olyan helyről jön egy kivétel, amire nem számítunk, például egy olyan virtuális függvényből, amit felülírtak, és nincsen ráhatásunk annak a kódjára

Időbeli csatolás

- a módosíthatóság problémája az is, hogy időbeli csatolást hozhat létre
- ez azt jelenti, hogy nem mindegy, hogy melyik függvényt melyik függvény után hívunk, mert a hívások más sorrendben történő intézésének más lesz az eredménye
- az alábbi példában a negyedik sor átkonfigurálja a request objektumot, és a második fetch hívás erre az átkonfigurálásra épít
- a probléma az, hogy az első fetch hívás az összekonfigurált állapotot újrahasznosítja, és ha az módosul, az kihatással lehet a második fetch hívásra is
- ráadásul a fetch hívásokat ebben a sorrendben kell intézni, nem lehet őket felcserélni, mert az teljesen más eredményhez vezetne (ezt jelenti az időbeli csatolás)

```
Request request = new Request("http://example.com");
request.method("POST");
String first = request.fetch();
request.body("text=hello"); // modifies the request
object String second = request.fetch();
```

Immutable objektumok előnyei

- könnyű létrehozni, tesztelni és használni
- mindig szálbiztos
- nincs szükség klónozásra és copy-konstuktorra
- mellékhatásmentesség
- identitás nem változik
- könnyű cache-elni
- nincs korrupt állapot
- megelőzhetők a null-referenciák
- elkerülhető az időbeli csatolás

Könnyű létrehozni, tesztelni és használni

- az immutable objektumok csak a konstruktoron keresztül befolyásolhatók, így a létrehozás nagyon egyszerű
- tesztelni is könnyű őket, mert ha két immutable objektumot ugyanúgy hozunk létre, akkor a viselkedésük is azonos lesz

Mindig szálbiztos

- mindig szálbiztosak, hiszen az immutable objektumok csak olvashatóak, mindegy, hogy hány szál éri el őket, minden szál ugyanazt az egy, konzisztens állapotot látja
- mivel az objektumok nem módosíthatók, a szálak nem tudják egymás állapotát elrontani

Nincs szükség klónozásra és copy-konstuktorra

- például egy üres immutable kollekció lehet singleton, amely nyugodtan beadható paraméterként, vagy visszaadható eredményként, senki sem fogja tudni az állapotát elrontani
- ez sokkal hatékonyabb, mintha minden egyes hívásnál egy módosítható kollekcióból újabb és újabb üres példányt kellene létrehozni

Mellékhatásmentesség

- az immutable objektumok mellékhatás mentesek
- bátran beadhatók egy függvénynek
- ha az a függvény bármilyen manipulációt szeretne végezni, az csak úgy megy, hogy új objektumokat hoz létre, az általunk beadott paraméterre ő nem tud hatással lenni
- ugyancsak nincsen szükség védelmi másolatokra, amikor a saját belső állapotunkat kiadjuk a külvilágnak
- itt látszik, hogy az immutable objektumok mennyivel hatékonyabbak tudnak lenni a módosítható társaiknál

Identitás nem változik

- mivel a belső állapotuk nem módosulhat, így az identitásuk sem változik
- ha egyszer egy immutable objektumot kulcsként használtunk, az kulcsként továbbra is működik, sőt, ha ugyanolyan paraméterekkel létrehozunk egy másikat, az ugyanolyan szerepkörben működhet, mint az eredeti, ugyanolyan kulcsként használható

Könnyű cash-elni

- Mivel az identitásuk nem változik, az immutable objektumokat könnyű cache-elni is az
- ha ugyanis ugyanolyan paraméterekkel hívnánk a konstruktort, akkor egy ekvivalens objektumhoz jutnánk, így, ha a paraméterek ugyanazok, akkor a cache-elt objektum is eljátszhatja ugyanezt a szerepet

Nincs korrupt állapot

- nem tud korrupt belső állapot létrejönni
- az ilyen objektumokat ugyanis a konstruktorban kell inicializálni, de hogyha ez a konstruktor kivételt dob, akkor létre sem jön az objektum

Megelőzhetők a null-referenciák

- az alapértelmezett objektumot, például egy kollekciónál az üres kollekciót definiálhatjuk Singleton-ként, és őt lehet használni null-referencia helyett
- ez hasznos lehet mind paraméter-átadáskor, mind pedig valamilyen értékkel való visszatéréskor, hogy null helyett az alapértelmezett objektumot használjuk
- így elkerülhetőek a nullpointer exception-ök, és betartható az a clean code szabály is, hogy ne adjunk be null értéket, és ne térjünk vissza null értékkel

Elkerülhető az időbeli csatolás

- az immutable objektumokkal az időbeli csatolás is elkerülhető
- az alábbi kódrészletben például az első és a második fetch hívás felcserélhető, a működés nem fog változni

```
final Request request = new Request("");
```

```
final Request post = request.method("POST");
String first = post.fetch();
String second = post.body("text=hello").fetch();
```

- ez annak köszönhető, hogy minden egyes „módosító” függvény valójában egy új objektumot hoz létre, amely egy új, belső állapottal rendelkezik, a meglévő objektumok pedig megmaradnak változatlanul
- így a függvényhívások nincsenek hatással más objektumokra, így pedig elkerülhető az időbeli csatolás

Immutable objektumok implementálása

Módosítható Date objektum implementálása

- kezdjünk egy módosítható Date objektummal, hogy legyen összehasonlítási alapunk
- az alábbi kódrészlet egy módosítható dátum implementációját mutatja, amely év, hónap és nap értékeket tud tárolni
- a property-knek vannak setterei is, tehát a dátum állapota (a dátum mezőinek értéke) ezeken keresztül kívülről bármikor módosítható

```
public class MutableDate {
    private int year;
    private int month;
    private int day;
    public MutableDate (int year = 0, int month = 0, int
    day = 0){
        this.year = year;
        this.month = month;
        this.day = day;
        }
        public int Year {
            get { return this.year; }
            set { this.year = value; }
        }
        public int Month {
            get { return this.month; }
            set { this.month = value; }
        }
        public int Day {
            get { return this.day; }
            set { this.day = value; }
    }
}
```

Immutable Date objektum implementálása

- az immutable dátum objektum implementálása kicsit hosszabb lesz, mert az imperatív nyelvek általában nem arra vannak felkészülve, hogy immutable objektumokkal dolgozzanak
- a következőkben az immutable dátum implementációjának egyes részein fogunk végigmenni

Immutable Date 1. rész – property-k, konstruktor

- az immutable dátumnál van egy Singleton az alapértelmezett dátum értékre, amely évként, hónapként és napként is a nulla értéket tárolja
- az év, hónap és nap mezők readonly-ként vannak definiálva
- ez főleg abban segít, hogy nehogy olyan kódot írjunk az osztályba, amely a konstruktoron kívül befolyásolni tudja ezek értékét
- a konstruktor inicializálja a mezőket
- a property-knek csak gettereik vannak, vagyis kívülről csak lekérdezhetőek a mezők értékei

```
public class Date {

// Immutable default value:
public static readonly Date Default = new Date();

// Fields are readonly to prevent accidental change:
private readonly int year;
private readonly int month;
private readonly int day;

public Date(int year = 0, int month = 0, int day = 0){
    this.year = year;
    this.month = month;
    this.day = day;
}
// Getters:
public int Year { get { return this.year; } }
public int Month { get { return this.month; } }
public int Day { get { return this.day; } }
//...
```

Immutable Date 2. rész - update függvények

- mivel az immutable objektumok belső állapota nem módosítható, szükség van olyan függvényekre, melyek segítségével új objektumok hozhatók létre egy meglévő objektum állapotából kiindulva
- ilyen lehet egy Update függvény, mely az összes mezőn tud dolgozni
- de mezőnként külön függvények is definiálhatók az év, hónap és nap "átírására"

```
public class Date{
    // ...
    // Update multiple fields:
    public Date Update(int year, int month, int day){
        if (year != this.year || month != this.month || day
        != this.day){
            return new Date(year, month, day);
        }
        return this;
    }
    // Update individual fields as a fluent API:
    public Date WithYear(int year){
```

```
        return this.Update(year, this.month, this.day);
    }
    public Date WithMonth(int month){
        return this.Update(this.year, month, this.day);
    }
    public Date WithDay(int day){
        return this.Update(this.year, this.month, day);
    }
    // ...
```

Immutable Date 3. rész - Builder-hez kapcsolódó függvények

- bizonyos immutable objektumok esetén szükség lehet egy Builder osztályra, amely az objektum módosítható változata, amely akkor lehet hasznos, ha az objektum felépítése sok-sok apró módosító műveletből áll, például egy lista esetén sok ezer elem közvetlen egymás után történő beszúrása
- mindenesetre most maradjunk ennél az egyszerű dátum példánál, de a megoldás természetesen általánosítható
- kell egy olyan függvény, amely Buildert készít az aktuális immutable objeketumból
- de kell egy statikus függvény is, amely egy alapértelmezett értékekkel kitöltött Buildert hoz létre

```
public class Date{
// ...
// Builder from the current object:
public Date.Builder ToBuilder(){
return new Date.Builder(this);
}
// Builder from the default value:
public static Date.Builder CreateBuilder(){
return new Date.Builder(Date.Default);
}
// ...
```

Immutable Date 4. rész - Builder mezők, konstruktor

- maga a Builder osztály nagyon hasonló egy hagyományos módosítható objektumhoz
- konstruktorában lemásolja a kapott immutable objektum állapotát, és ez lesz a Builder kezdőállapota

```
public class Date{
    // ...
    // Inner class of Date:
    public class Builder {
        private int year;
        private int month;
        private int day;

        internal Builder(Date date) {
            this.year = date.Year;
            this.month = date.Month;
            this.day = date.Day;
```

```
        }
        // ...
```

Immutable Date 5. rész - Builder getterek, setterek

- a Builder következő része a hagyományos property-k, amelyek getterekkel és setterekkel is rendelkeznek, hiszen a Builder egy normál, módosítható objektum

```
public class Date {
    // ...
    public class Builder {
        // ...
        // Getters-setters:
        public int Year {
            get { return this.year; }
            set { this.year = value; }
        }
        public int Month {
            get { return this.month; }
            set { this.month = value; }
        }
        public int Day {
            get { return this.day; }
            set { this.day = value; }
        }
        //...
```

Immutable Date 6. rész - Builder setterek fluent API-hoz, ToImmutable()

- a Builder, ahogy nevében is benne van, a Builder tervezési mintát követi, így ő is rendelkezhet olyan függvényekkel, amelyek segítségével a Builder fluent API-ként használható
- végezetül a Buildernek szükdége van egy olyan függvényre, amely segítségével a Builder aktuális állapotából egy immutable objektum nyerhető ki

```
public class Date {
    // ...
    public class Builder {
        // ...
        // Setters for the fluent API:
        public Builder WithYear(int year){
            this.Year = year;
            return this;
        }
        public Builder WithMonth(int month){
            this.Month = month;
            return this;
        }
        public Builder WithDay(int day){
            this.Day = day;
            return this;
        }
```

```
            // Construct an immutable object
            //from the current state of the builder:
            public Date ToImmutable(){
                return new Date(this.year,
                    this.month, this.day);
            }
        }
    }
```

Immutable Date objektum használata

- látható, hogy az immutable változat egy kicsit kevésbé kényelmes, mint egy módosítható dátum
- ennek főként az az oka, hogy az imperatív nyelvek a módosítható objektumokra lettek kitalálva, az immutable objektumok számára kissé kényelmetlenek
- mindenesetre a Builder változat jobban hasonlít a módosítható objektumokra, így az valamivel kényelmesebben használható egy immutable objektum összerakására

```
Date d1 = Date.Default; // 0.0.0.
Date d2 = d1.WithYear(2016); // 2016.0.0.
Date d3 = d1.WithYear(2017).WithMonth(9); // 2017.9.0.
Date d4 = d3.Update(d3.Year, 5, 23); // 2017.5.23.

Date.Builder b1 = Date.CreateBuilder(); // 0.0.0.
b1.Year = 2016;
b1.Month = 9;
Date d5 = b1.ToImmutable(); // 2016.9.0.
b1.Day = 27;
Date d6 = b1.ToImmutable(); // 2016.9.27.

Date.Builder b2 = d4.ToBuilder(); // 2017.5.23.
b2.Month = 3;
b2.Day = 21;
Date d7 = b2.ToImmutable(); // 2017.3.21.

// 2017.6.18.
Date d8 =
d7.ToBuilder().WithMonth(6).WithDay(18).ToImmutable();
```

Immutable Date objektumot tartalmazó immutable osztály

- készítsünk egy olyan immutable Person osztályt, melynek születési dátuma egy ilyen immutable date osztály
- elsőre sok újdonság nincsen, az immutable személy megvalósítása hasonló mintát követ, mint az immutable dátum
- az egyetlen fő különbség a személy Builder osztályában van, mert annak születési dátuma dátum Builder típusú kell, hogy legyen, és így, amikor a PersonBuilder-t létrehozzuk, a születési dátumból is Builder-t kell készíteni
- amikor pedig a Builder-t visszaalakítjuk immutable személlyé, akkor a dátutmot is vissza kell alakítani immutable dátummá

- látszik, hogy ez a megoldás nem túlságosan szép, és nem is túlságosan hatékony
- éppen ezért nem mindig éri meg az immutable osztály Builder változatának elkészítése

```
// Immutable date to date builder:
this.birthDate = person.BirthDate.ToBuilder();

// Date builder to immutable date:
return new Person(this.name, this.height,
this.birthDate.ToImmutable());
```

Immutable objektumok hátrányai

- túl sok kód
- kényelmetlen szintaxis
- nem lehet körkörös referencia
- változásokkal kapcsolatos hátrányok

Túl sok kód

- láttuk, hogy milyen sok kódot igényel egy immutable osztály és annak Builder változata
- a sima, módosítható dátum 25 sorral elintézhető, míg annak immutable változata a Builderrel együtt körülbelül 100 sorból oldható meg
- ez azt jelenti, hogy körülbelül 4x annyi kódot kell írni az immutable objektumukhoz, mint a hagyományos, módosítható változatukhoz
- ennek mindössze csak az az oka, hogy az imperatív programozási nyelvek nem erre lettek kitalálva
- mivel azonban a szükséges kód elég manuális előállítható, akár programkód automatikus generálásával is előállíthatók lehetnek az immutable osztályok forráskódjai

Kényelmetlen szintaxis

- a sok kódra azért van szükség, hogy imperatív nyelvekben kívülről kényelmesebben lehessen használni az immutable objektumokat
- sajnos azonban az immutable változatok kevésbé intuitív módon használhatóak mint a módosítható változatok, de azért a kód nem megy az olvashatóság rovására

Nem lehet körkörös referencia

- az immutable objektumok következő hátránya, hogy nem lehet közöttük körkörös referenciát létrehozni
- például a férj és feleség nem köthető össze, mert valamelyiket hamarabb kell létrehozni, de annak konstruktorába nem lehet beadni paraméterként a másikat, aki később jön létre
- hogyha az ember jobban belegondol, ez a kijelentés nem állja meg teljesen a helyét, hogy nem lehet körkörös referenciát létre hozni
- az immutable objektumoknak ugyanis lehetnek lusta kiértékelésű attribútumai
    - hogyha ügyesek vagyunk, akkor ezek segítségével megoldhatók a körkörös referenciák

- egy másik lehetséges megoldás lehet az is, hogy nem a férj és feleség tárolják egymásról, hogy kihez tartoznak, hanem van egy házasság objektum, vagyis a férj és feleség közötti asszociációt egy külön immutable objektum reprezentálja

Változások

- az immutable objektumok további hátrányai a változtatásokkal kapcsolatosak
- imperatív nyelvekben olcsóbb egy létező objektumot módosítani, mint egy új objektumot létrehozni
- ugyancsak kényelmetlen lehet egy heterogén immutable struktúrában nagyon mélyen apró változtatás végrehajtása
- gondoljunk bele például, hogyha egy world dokumentumot gépelés közben karakterenként újra és újra fel kellene építeni, programkódban az elég kényelmetlen, és nem is túlságosan hatékony, bár nyilvánvalóan a dokumentum nem változó részei teljes mértékben újrahasznosíthatók
- azonban egy ilyen megoldás sem példátlan, amióta a C# kódot a Roslyn fordító fordítja, a Visual Studió minden egyes karakter begépelésekor újra és újra felépíti a teljes solution immutable modelljét (természetesen a nem változó részeket újrahasznosítva)
- ez a példa bizonyítja azt, hogy ez a hátrány nem feltétlenül megy a teljesítmény rovására
- a változásokkal kapcsolatos hátrányok közül a harmadik, hogy az immutable objektum és a builder változat közötti konverzió nem igazán hatékony
- a kettő közötti konverzió túl sok másolást igényel, így csak akkor éri meg csinálni, ha a Builder-rel nagyon sok műveletet végzünk, és azoknak a műveleteknek a hatékonysága ellensúlyozni tudja az immutable objektum és a Builder közötti konverziót

Immutable gyűjtemények .NET-ben

Struktúra típusú gyűjtemény:

- ImmutableArray<T>

Osztály típusú gyűjtemény:

- ImmutableStack<T>
- ImmutableQueue<T>
- ImmutableList<T>
- ImmutableHashSet<T>
- ImmutableSortedSet<T>
- ImmutableDictionary<K, V>
- ImmutableSortedDictionary<K, V>

ImmutableArray<T>

- az elemek hozzáadásakor mindig eggyel nő a mérete tömbnek, és a régi elemek lemásolódnak
- nyilvánvalóan ez nem egy túl hatékony megoldás, éppen ezért az ajánlás az, hogy ImmutableArray-t csak akkor használjunk, hogyha az elemek száma nagyon kevés (kevesebb, mint 16), vagy a tömböt arra használjuk, hogy sok elemet tároljunk, de ritkán adunk hozzá új elemeket

ImmutableList<T>

- az immutable lista belül kiegyensúlyozott bináris fákra épül, így az új elem hozzáadása, a beszúrás, a törlés és az indexelés is logaritmikus idejű
- éppen ezért az immutable listát érdemes használni az immutable tömb helyett akkor, hogyha sok elemünk van, és sokat kell beszúrni és törölni
- az immutable lista viszont egyáltalán nem hatékony, ha be kell járni az elemeket
- ha az elemek ritkán cserélődnek, de sokszor kell bejárni őket, akkor mindenképpen az immutable tömb használata javasolt

Immutable kollekciók használata

- üres kollekció
  - immutable kollekciókat nem kell önállóan konstruktorral létrehozni, kiindulásként használható az üres, vagyis empty változatuk

```
var emptyFruitBasket = ImmutableList.Empty;
```

- elem hozzáadása
  - egy új elem hozzáadásakor egy újabb kollekciót kapunk, a régi nyilván megmarad változatlanul
  - vigyázzunk azonban, hogy ne felejtsük el egy változóban elmenteni a hozzáadás során létrejövő új kollekciót, különben a hozzáadás műveletünk hatástalan lesz
  - a helyes megoldás tehát az, hogy az Add függvény visszatérési értékeként kapott kollekciót elmentjük egy változóba

```
var  = emptyFruitBasket.Add("Apple");
```

- több elem hozzáadása
  - az értékadások láncolhatóak is, hogyha több elemet is szeretnénk még hozzáadni a kollekcióhoz

```
var fruitBasket = ImmutableList.Empty;
fruitBasket = fruitBasket.Add("Apple");
fruitBasket = fruitBasket.Add("Banana");
fruitBasket = fruitBasket.Add("Celery");
```

  - hogyha egyszerre több elemet szeretnénk belerakni a kollekcióba, azt az AddRange függvénnyel megtehetjük, amely bizonyos esetekben hatékonyabb is lehet, mint az elemek egyenkénti hozzáadása

```
fruitBasket = fruitBasket.AddRange(new[] {
  "Kiwi", "Lemons", "Grapes" });
```

  - a hívások természetesen közvetlenül is láncolhatók, és elég a végeredményt elmenteni egy változóba

```
var fruitBasket = ImmutableList.Empty
.Add("Apple")
.Add("Banana")
.Add("Celery")
.AddRange(new[] { "Kiwi", "Lemons", "Grapes" });
```

- Gyűjtemény Builder-ek
  - a gyűjteményeknél, kifejezetten akkor, hogyha sok elemről van szó, érdemes áttérni a Builder változatra, és a Builder változatot manipulálni, majd az eredményt újra visszaalakítani immutable-re
  - hogyha sok elemről van szó, ez sokkal hatékonyabb tud lenni, mintha az immutable változaton végeznénk el műveleteket
- Kötegelt módosítás

- a gyűjtemények rendelkeznek kötegelt műveletekkel is, amelyek ugyancsak hatékonyabbak tudnak lenni, mint a kollekció kézzel való manipulálása, hiszen ezek a háttérben át tudnak térni Builderre, és ott hatékonyabban elvégezni a műveletet

```
private ImmutableHashSet<Customer> customers;
public ImmutableHashSet<Customer> GetCustomersInDebt(){
    return this.customers.Except(c => c.Balance>=0.0);
}
```

Immutable vs. mutable gyűjtemények

Immutable vs. mutable gyűjtemények

- bizonyos esetekben az immutable gyűjtemények hatékonyabbak is tudnak lenni, mint a módosítható változataik
- különösen igaz ez akkor, amikor a módosítható gyűjteményekből védelmi másolatokkal kell visszatérni

Immutable vs. mutable gyűjtemények komplexitása

	Mutable (átlagos)	Mutable (legrosszabb)	Immutable
Stack.Push	O(1)	O(n)	O(1)
Queue.Enqueue	O(1)	O(n)	O(1)
List.Add	O(1)	O(n)	O(log n)
ImmutableArray.Add			O(n)
HashSet.Add	O(1)	O(n)	O(log n)
SortedSet.Add	O(log n)	O(n)	O(log n)
Dictionary.Add	O(1)	O(n)	O(log n)
SortedDictionary.Add	O(log n)	O(n log n)	O(log n)

Immutability vs. mutability tanulság

- tanulságként azt mondhatjuk, hogy használjunk immutable objektumokat amikor csak lehet
- a nem módosíthatóság számos bug-tól és szálkezelési problémától meg tud minket óvni
- módosítható objektumok akkor előnyösek, hogyha nagyon sok, és nagyon gyakori változtatásra van szükség
- de a módosítható objektumokat sokkal nehezebb többszálú környezetben biztonságosan használni